Low-Cost Soft Error Resilience with Unified Data Verification and Fine-Grained Recovery for Acoustic Sensor Based Detection

Qingrui Liu*, Changhee Jung*, Dongyoon Lee* and Devesh Tiwari†
* Virginia Tech, Blacksburg, Virginia, USA
†Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA
Email: lqingrui@vt.edu, chjung@cs.vt.edu, dongyoon@vt.edu, tiwari@ornl.gov

***Abstract*—This paper presents Turnstile, a hardware/software cooperative technique for low-cost soft error resilience. Leveraging the recent advance of acoustic sensor based soft error detection, Turnstile achieves guaranteed recovery by taking into account the bounded detection latency. The compiler forms verifiable regions and selectively inserts store instructions to checkpoint their register inputs so that Turnstile can verify the register/memory states with regard to a region boundary in a unified way without expensive register file protection.**

At runtime, for each region, Turnstile regards any stores (to both memory and register checkpoints) as unverified, and thus holds them in a store queue until the region ends and spends the time of the error detection latency. If no error is detected during the time, the verified stores are merged into memory systems, and registers are checkpointed. When all the stores including checkpointing stores prior to a region boundary are verified, the architectural and memory states with regard to the boundary are verified, thus it can serve as a recovery point. In this way, Turnstile contains the errors within the core without extra memory buffering.

When an error is detected, Turnstile invalidates unverified entries in the store queue and restores the checkpointed register values to get the architectural and memory states back to what they were at the most recently verified region boundary. Then, Turnstile simply redirects program control to the verified region boundary and continues execution. The experimental results demonstrate that Turnstile can offer guaranteed soft error recovery with low performance overhead (<8% on average).

***Keywords*—Soft Error Resilience, Fine-Grained Recovery, Acoustic Sensor, Compiler, Region Boundary Buffer**

I. Introduction

Due to technology scaling and near-threshold computing, radiation-induced soft errors become a challenging concern in computing systems [1], [2], [3]. When high-energy particles (e.g., alpha or cosmic neutron particles) strike the circuit, they might lead to an application crash or even worse, silent data corruption (SDC) where the errors corrupt the program output without being detected. The near-threshold voltage and the process variation make it harder to predict the response of the circuit to a particle strike, and thus they become significantly more susceptible to soft errors [1], [4], [5], [6], [7].

For soft error resilience, detection and recovery are two essential steps to expose the errors and fix any resulting corruption. Recently, Upasani et al. have proposed an acoustic sensor based detection [3], [8], [9] as an alternative to traditional redundancy or symptom-based schemes [10], [11], [12], [13], [14], [15]. Acoustic sensors are very promising in that they can detect soft errors within the core by sensing the sound wave made by actual particle strikes with bounded detection latency (<30 cycles) at the cost of $\sim$1% area overhead [3].

Since every particle strike detected by the acoustic sensors does not necessarily produce an error in the output of the program [16], [17], fine-grained recovery schemes with short rollback distance (<100 instructions) are more favorable. The reason is that fine-grained recovery schemes can tolerate high false positives since they only need to rollback beyond a small amount of instructions. However, prior fine-grained recovery schemes [18], [19], [20], [21], [22] cannot provide core-level error containment which in turn requires an extra memory buffer. Besides, prior fine-grained recovery schemes incur expensive hardware and performance overhead to preserve the architectural and memory states of the recovery points. More importantly, simply combining acoustic sensors-based detection and prior fine-grained recovery schemes cannot guarantee to recover the system due to the detection latency of acoustic sensors [21] as Section II-B shows.

To this end, this paper presents Turnstile, a lightweight hardware/software cooperative technique for soft error resilience that leverages the acoustic sensors. The design objective of Turnstile is to provide low-cost and fine-grained checkpoint/rollback/re-execution mechanisms. Turnstile is motivated by the insight that all the committed instructions before a program point p are verified only if the acoustic sensors raise no alarm during the time of the error detection latency since p. Therefore, Turnstile partitions the program into different verifiable fine-grained regions and proposes a simple hardware support to verify those regions one by one, preventing any errors from escaping the core. In case of an error, Turnstile rollbacks the program to the most recently verified region boundary rb, restores rb's architectural and memory states, and re-executes from rb to recover from the error.

For the region based verification and recovery, Turnstile needs to verify and preserve the program state (registers and memory) with regard to the beginning of each region (i.e., rb). As for the memory state, Turnstile leverages gated store queue (GSQ) to buffer all the committed yet unverified stores before

978-1-5090-3508-3/16/$31.00 © 2016 IEEE

rb until its last preceding region (i.e., the region before rb) ends and spends the time of the error detection latency [23], [24], [25]. Once the last preceding region is verified, its committed stores can be drained to the L1 cache [1]. If a fault happens since then, all the unverified stores after rb are flushed out (squashed) from the GSQ, thus the memory state with regard to rb is always preserved.

As for the register state, Turnstile proposes unified data verification which leverages a novel compiler analysis to automatically identify the minimal register state necessary for restoring a recovery point in case of a fault, and inserts stores to checkpoint their value in a reserved checkpoint location of the memory. This approach allows Turnstile to convert the problem of register data verification into that of memory data verification, which is very cheap since stores are not on the critical path in general. Therefore, the stored data of the not-yet-verified registers are eventually verified in the same way of memory data verification. The error recovery process remains the same as well, i.e., reading from verified (checkpointed) register data from memory.

In particular, to realize the proposed verification and recovery, Turnstile introduces a simple hardware support called RBB (region boundary buffer) that is off the critical path of the pipeline and interacts with the ROB (reorder buffer) and the GSQ (gated store queue). Taking into account the error detection latency, the RBB precisely controls the GSQ for timely verification as well as directs the program control to the most recently verified recovery point on a fault. The ROB notifies the RBB of each committed recovery point, and the RBB notifies the GSQ of unverified stores to be squashed during the error recovery.

Finally, for the proposed hardware support, Turnstile introduces another new compiler analysis to statically partition the whole program into different verifiable fine-grained regions considering the capacity of the GSQ in the core. For Turnstile to buffer all the stores in an unverified region, its compiler guarantees that the number of stores in a single region never exceeds the size of the GSQ to contain the errors within the core. Besides, to avoid performance degradation, Turnstile forms the regions such that the stores in the neighboring regions can be verified in a pipelined fashion, i.e., overlapping the verification of one region with the execution of the next region.

The following are the contributions of this work:

- A low-cost soft error resilient solution that offers guaranteed, fine-grained recovery with core-level error containment. Turnstile does not require expensive hardware support (e.g., large store buffer and ECC-protected register file) for recovery. The runtime overhead is only $\sim$8% on average.
- A new hardware support called RBB that interacts with GSQ and ROB to efficiently and precisely realize the region verification and the recovery considering the latency of the sensor-based soft error detection.
- A novel compiler analysis framework, that requires no source code, to automatically partition the whole program into verifiable fine-grained regions taking into account Turnstile's hardware support, as well as to checkpoint the minimal architectural state with regard to the region boundaries.

[1]Stores are merged to L1 when the bandwidth of the bus between the GSQ and the cache is available

II. Background and Challenges

This section discusses the state-of-the-art soft error detection/recovery schemes and their limitations. Throughout the paper, a ***region*** will refer to a single-entry, multiple-exits subgraph of control flow graph where the entry block dominates all other blocks [26]. This paper also uses the term ***inputs*** to refer to the variables live-in to the region boundary.

A. Sensor-Based Soft Error Detection

Recently, researchers have proposed a lightweight approach that detects the actual particle strike rather than its impact to the program execution [8], [3], [27], [28], [29]. When a particle collides with a silicon nucleus, the ionization process produces a mass of electron-hole pairs, which subsequently generate phonons and photons resulting into an acoustic wave. Thus, the particle strike can be detected by the change in the capacitance of a cantilever beam like sensor [8], which is placed on top of the processor. It can detect a strike that is 5mm away from the detector within 500ns (1000 cycles at 2 GHz) [8].

The detection latency can be bounded by both the number of sensors deployed and their elaborate placement, and is referred to worst case detection latency (WCDL). For example, Upasani *et al.* [3] proposed to leverage 300 detectors, which results in 30 cycles WCDL at 2GHz at the cost of $\sim$1% area overhead.

Turnstile assumes this kind of soft error detection scheme with a WCDL and focuses on designing detection latency aware recovery mechanism. Note that the prior work used the majority of the detectors to cover large storage units such as caches. Given that the caches have already been protected with ECC [30], [31] in the commodity processors (e.g., ARM Cortex-A57 [32]), we expect that the detection latency can be further decreased with the same number of detectors when they are used to protect only smaller portion of processing units such as RF, RAT, PC, and ALU, which is sufficient for Turnstile as it contains soft errors within the core (i.e., the effects of soft errors do not escape to the memory system).

B. Fine-grained Soft Error Recovery and Challenges

State-of-the-art fine-grained recovery schemes [22], [18], [19], [20], [21], [33], [34] follow the checkpoint, rollback and re-execution model, but at a much finer granularity (<30 instructions on average [19]). They divide the control flow graph into different regions and correct a soft error by jumping back to the beginning of the faulty region. To achieve this, the compiler guarantees the inputs to the region boundary are not overwritten, i.e., no anti-dependence (write-after-read) on

the inputs, during the execution of the region. That way the region inputs remain the same within the region, making the regions harmless to be re-executed multiple times. If some inputs are overwritten within the region, their values do not remain the same as they were at the region boundary making the re-execution of the region unsafe, i.e., ending up with unexpected output. Thus, prior fine-grained recovery schemes require that the region inputs are never overwritten during the execution of the region. For example, De Kruijf *et al.* place region boundaries to break the memory-level anti-dependence and leverage register renaming to eliminate the register anti-dependence on the inputs to the region [19], [20], [21].

Despite the benefit of false-positive tolerance and region-level error containment, prior fine-grained recovery schemes expose three critical challenges that hinder their pervasive use.

- *Challenge 1*: Prior fine-grained recovery schemes contain the error within the region rather than the core. If the destination address of a store is corrupted and the store is performed, the memory state might be corrupted. Therefore, those recovery schemes assume a large store buffer to hold all the stores in a region. Besides, prior region partition algorithms [19], [18] fail to constrain the number of stores in a region which might overflow the store buffer leaving the possibility of failure to recovery.
- *Challenge 2*: Prior fine-grained recovery schemes have to pay a high overhead to protect the region inputs from being corrupted (overwritten) by the soft errors (anti-dependence). For example, [19] needs to protect the register files (RF) with error correcting code (ECC) as well as hardening logic for RF's controller which is on the processor's critical path. Besides, [19] leverages register renaming to eliminate the register anti-dependence which leads to performance degradation. Even worse, [18] gives up the protection of some regions due to the high overhead caused by preserving all the their inputs.
- *Challenge 3*: An error must be detected within the region, which either requires an expensive detector to ensure the correctness of region inputs or sacrifices error coverage. If an error occurring in one region is detected in the next region, simply re-executing it cannot achieve the recovery since its inputs may have been corrupted by the error. In general, a majority of regions in the state-of-the-art schemes are very short (e.g., $\approx$ 10 instructions [19], [21]) which requires an expensive error detector with almost zero latency. Thus, acoustic sensors cannot work with prior fine-grained recovery schemes.

The following sections show how Turnstile differs from the previous fine-grained recovery schemes and solves the above challenges at a low cost.

III. Turnstile Overview

The goal of Turnstile is to provide a low-cost fine-grained hardware/software cooperative technique for soft error resilience with guaranteed error recovery which can work with detectors with detection latency, e.g., acoustic sensors.

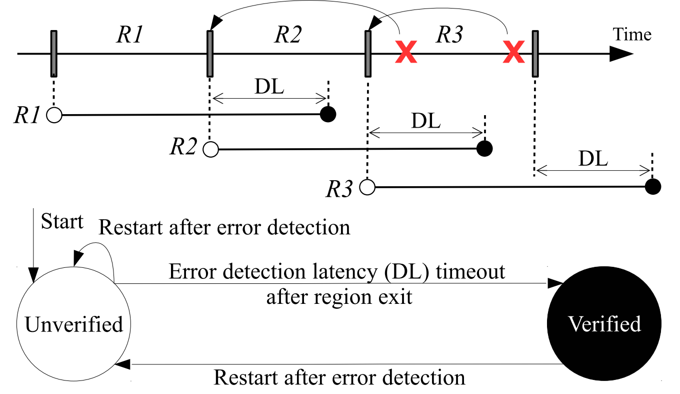

Fig. 1. The region verification/recovery idea (top) and the status change of region execution (bottom)

*a) **Containing The Errors Within The Core**:* Turnstile proposes a software/hardware co-design scheme to contain the errors within the core. Turnstile first partitions the entire program into different fine-grained regions preventing the number of stores in each region from overflowing the gated store queue (GSQ) (Section IV). Then, Turnstile introduces the region boundary buffer (RBB) to control the GSQ to hold the stores of a region until the region ends and spends WCDL cycles (Section V). Thus, Turnstile can verify the entire region as fault-free after WCDL cycles, by carefully exploiting the acoustic sensor-based detector and the GSQ. Figure 1 describes the status change of each region during its execution in the presence of soft errors. Initially, Turnstile regards all the committed stores in the region as unverified, thus holding their write-back to memory until the region is verified, i.e., if there is no error detected during the time of WCDL after the region exit. For regions R1 and R2, they can drain the stores in their region only after the time of WCDL elapses from when they reach the next region boundaries, i.e., R2 and R3, respectively. Consequently, no faulty store of unverified regions can be drained to memory system (L1 cache), That is, Turnstile can contain the error occurred in a region within the core.

*b) **Unified Data Verification**:* Premised on Turnstile's hardware support (Section V), Turnstile enables unified data verification. Each region generates some or all of the inputs to the later regions by defining the variables that can reach the later regions. The region input is two-fold: (1) the values stored in memory, simply memory inputs. They can be verified by leveraging the Turnstile's RBB and GSQ; and (2) the values of register, i.e., register inputs. Turnstile leverages a novel compiler analysis [34] to insert checkpoints right after the instructions that define the registers as long as they are used as the inputs of later regions (Section IV). Note that the checkpoints are essentially store instructions that save the checkpointed register value in reserved memory location. In effect, Turnstile transforms the verification of register region inputs into that of their store instructions for checkpointing. That way Turnstile can unify the verifications of memory and register inputs.

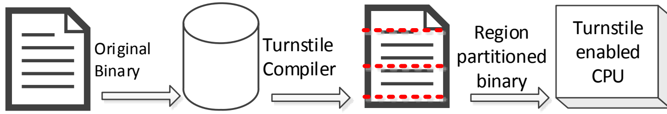

Fig. 2. The high-level view of Turnstile

*c) **Detection Latency Aware Rollback and Re-execution Based Recovery**:* Turnstile can successfully recover the system even if the error is not detected within the faulty region, i.e. Turnstile can tolerate the error detection latency of acoustic sensors. If an error is detected during the time of WCDL after a region (R2) encounters its next region boundary (R3), e.g., the first error in R3, Turnstile's recovery handler takes the following three actions to recover the error:

- First, Turnstile discards all the unverified stores in the GSQ, i.e., all the stores and checkpointing stores after R2's entry point. Note that all the entries in the internal structures (e.g. instruction queue etc.) are invalidated and drained from the pipeline as well.
- Second, it restores the region input of R2 next to most recently verified region R1. Since all the stores after region R2's entry are discarded, the entire memory remains intact, that includes the memory inputs to R2 as well as the reserved memory location for the register checkpointing. Turnstile can thus safely restore the register inputs to R2 by loading the values from the location.
- Finally, Turnstile jumps back to R2's entry and continue execution from there to correct the error.

As shown in Figure 1, the following regions (R2, R3) are then executed and verified once again. Note that to recover from the second error detected in R3, Turnstile redirects the program control to the beginning of R3, since R2 has already been verified.

*d) **Addressing the Previous Challenges**:* Turnstile solves the three challenges in the previous section. *Challenge 1* is solved with Turnstile's compiler and hardware support to contain the error within the core without sacrificing error recovery capability. *Challenge 2* is also solved with unified data verification. Turnstile does not require any hardware protection for register files since it handles the register verification in a unified way as discussed above. Besides, as the checkpointing stores are generally off-the-critical-path of the processor, we expect the performance degradation is small. *Challenge 3* is solved, since Turnstile does not require an error to be detected within the region. This is because Turnstile can always guarantee to recover the inputs of the most recently verified region boundary, i.e., the inputs to the faulty region.

Figure 2 shows a high-level view of Turnstile. An application binary is fed into the compiler, in this case a binary rewriter. The resulting binary has special instructions to represent a region boundary as well as store instructions to checkpoint registers. During program execution, Turnstile holds all the unverified data stored in a region using a gated store queue in the first place. At the same time, the verification controller of the Turnstile processor verifies each region execution by intelligently counting down the detection latency time after the region exit. Once a region is verified, the controller signals the GSQ to release those data stored by the region, and keeps the PC of the first instruction of the next region. For later errors, Turnstile uses the PC as a recover point once flushing (squashing) every unverified data in the GSQ and restoring the region inputs.

IV. TURNSTILE COMPILER

Turnstile compiler primarily performs two tasks on the binary before it runs on the Turnstile-enabled processor. First, Turnstile partitions the binary into different verifiable fine-grained regions considering the capacity of the gated store queue (GSQ) in the processor to contain the error within the core. Then, the compiler inserts checkpointing store instructions to preserve the register inputs to the regions so that Turnstile can achieve unified data verification with simple hardware support called RBB (region boundary buffer) shown in Section V that controls and interacts with the GSQ and the ROB (reorder buffer).

As Turnstile's region partition algorithm requires the knowledge of checkpoints to be inserted, we first introduce the checkpoint set identification in Section IV-A before discussing the region partition algorithm in Section IV-B. Finally, we present our loop optimization technique in Section IV-C.

A. Checkpoint Set Identification

Turnstile leverages a novel compiler analysis to identify the minimal register state [34] necessary for restoring a recovery point (the most recently verified region boundary) in case of a fault. Then, Turnstile inserts stores to checkpoint those register value in the reserved checkpoint locations in the memory. The checkpoint-set analysis investigates register-updating instructions to checkpoint the resulting value being used in later regions as their *live-in* registers [26]. That is, Turnstile is interested in the last among those instructions that update the same register. For a given region r, Turnstile thus cares about only the instruction d whose register definition is *downward-exposed* [26]. To determine if the updated register needs to be checkpointed, Turnstile evaluates the following boolean function.

$$Ckpt(r,d) = \begin{cases} 1 & \text{if } (Reg(d) \in LiveOut(r)) \\ 0 & otherwise \end{cases}$$

where Reg maps an instruction to its first operand (i.e., the destination register) while $LiveOut$ maps a region to a set of *live* registers at the end of the region. If the output of $Ckpt(r,d)$ is set, Turnstile simply inserts a checkpoint (i.e., a store instruction) immediately following those instructions d to store the updated value $Reg(d)$ in a reserved memory location. Note that for each register being checkpointed, Turnstile reserves a specific region in the stack frame. Thus, the worst-case memory overhead due to the checkpoints is bound to the register file size times the maximum stack depth at runtime.

B. Region Formation

At a first glance, forming regions appears to be as simple as counting the store instructions while traversing the control flow graph (CFG) and placing boundaries whenever a threshold (i.e., half size of GSQ) is reached. However, before determining the region boundaries, Turnstile cannot determine the locations where checkpoints (i.e., stores) are inserted, making the region formation non-trivial. That is, the region formation problem is circularly dependent on itself: determining region boundaries requires the resulting instrumented binary containing checkpoints, which in turn requires identification of the regions.

To solve the problem, Turnstile first considers all the basic blocks in the CFG as initial regions, and calculates the number of checkpoints to be instrumented in each region. Traversing the CFG, Turnstile then attempts to combine those initial regions into larger regions as much as possible. By combining them, Turnstile can eliminate many checkpoint instructions (i.e., stores). This is because the registers being checkpointed are no longer live-out to the later regions. The key of our heuristic is to ensure the number of stores during the execution of each region will not overflow the threshold (i.e., half size of GSQ) even after the checkpointing stores are inserted. Note that we set our threshold to be half the size of the store queue so that Turnstile can overlap the verification of one region (i.e., waiting for WCDL at the end of the region) with the execution of the next region. Therefore, Turnstile can greatly reduce the pipeline stalls due to the store queue overflow.

Algorithm 1 Turnstile Region Formation Algorithm

1: Place a boundary at the beginning of functions and at the end of each callsite.
2: Place a boundary at the beginning of the loop header of each loop.
3: Place a boundary at a memory fence and an atomic operation.
4: **for** each basic block bb_i in CFG **do**
5: $\quad Str_{bb_i} \leftarrow Str_ori_{bb_i} + Str_ckpt_{bb_i}$
6: $\quad IncomeStr_{bb_i} \leftarrow 0$
7: **end for**
8: **for** each basic block bb_i in program topological order **do**
9: $\quad$ **if** bb_i starts with region boundary **then**
10: $\quad\quad$ accum_str $\leftarrow Str_{bb_i}$
11: $\quad$ **else**
12: $\quad\quad$ accum_str $\leftarrow Str_{bb_i} + IncomeStr_{bb_i}$
13: $\quad$ **end if**
14: $\quad$ **while** accum_str $>$ threshold **do**
15: $\quad\quad$ place boundary and split bb_i into $bb_i{}'$ and bb_i
16: $\quad\quad$ recalculate $Str_ori_{bb_i}$ and $Str_ckpt_{bb_i}$
17: $\quad\quad$ accum_str $\leftarrow Str_ori_{bb_i} + Str_ckpt_{bb_i}$
18: $\quad$ **end while**
19: $\quad$ **for** each successor basic block bb_j of bb_i **do**
20: $\quad\quad IncomeStr_{bb_j} \leftarrow$ max($IncomeStr_{bb_j}$, accum_str)
21: $\quad$ **end for**
22: **end for**

Algorithm 1 shows the heuristic. First, Turnstile considers all the entry and exit points of functions as region boundaries (line 1), as prior works do [19], [20], [21]. Second, Turnstile places a boundary at the beginning of each loop header (line 2) [2]. Third, Turnstile treats memory fences and atomic operations as region boundaries (line 3). The ISA enforces an ordering constraint on memory operations before and after the memory fences (e.g, x86 TSO consistency model requires that queued stores must be merged into the memory system at a fence instruction). In addition, atomic operations (e.g., atomic compare-and-swap) should complete, i.e., obtain the write permission and become visible to other processors [3]. As these are critical to guarantee correctness of synchronization operations for multithreaded programs, Turnstile places a boundary at a memory fence and an atomic operation, similar to [19]. Then, Turnstile identifies the basic block that has region boundaries in the middle of it, and splits it into different basic blocks. This guarantees that region boundaries always start at the beginning of basic blocks, thus helping the next step to compute the initial checkpoint instructions.

After placing the first set of region boundaries, Turnstile further analyzes the program and, if necessary, places additional boundaries to prevent the GSQ from overflowing. In line 4∼7, as each basic block bb_i is initially regarded as a region, Turnstile computes the total number of stores (Str_{bb_i}) in bb_i as the sum of original stores ($Str_ori_{bb_i}$) and checkpointing stores ($Str_ckpt_{bb_i}$).

Based on the checkpoint set identification (Section IV-A), Turnstile conservatively calculates the number of checkpointing stores in each basic block bb_i with the following equation:

$$Str_ckpt_{bb_i} = Def_{bb_i} \bigcap LiveOut_{bb_i} \tag{1}$$

where Def_{bb_i} is the set of registers defined in bb_i and $LiveOut_{bb_i}$ is the set of live-out registers of bb_i. Then, Turnstile assigns zero to $IncomeStr_{bb_i}$ which will be updated with the maximum of number of incoming stores accumulated from bb_i's predecessors during the later analyses.

And then, Turnstile traverses the CFG in topological order, consulting Str_{bb_i}, i.e., the total number of stores of each basic block bb_i (line 8∼22). In line 9∼13, if bb_i already has a region boundary at the entry, Turnstile updates the accumulating store number (accum_str) with Str_{bb_i} since we have started a new region. Otherwise, Turnstile keeps combining each basic block, updating accum_str with the sum of Str_{bb_i} and $IncomeStr_{bb_i}$, i.e., the incoming store number of bb_i. If accum_str is greater than the threshold which is half the size of the GSQ (line 14∼18), Turnstile tries to place a new boundary after a store where the number of stores in the region prior to the new boundary is less than

[2] Since the loop trip count is not always statically known, GSQ may overflow in some number of iterations even with a single store in the loop body. Section IV-C proposes our optimization to remove such a mandatory region boundary for certain loops.

[3] TSO model also requires draining the store queue at an atomic operation as it should ensure the program order between stores, whereas RMO model only requires the store of the atomic operation to be completed as it may employ unordered store queue [35]

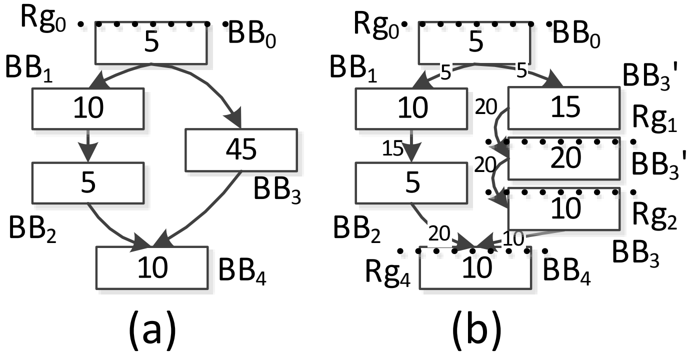

Fig. 3. An example of Turnstile Region Partition Heuristic

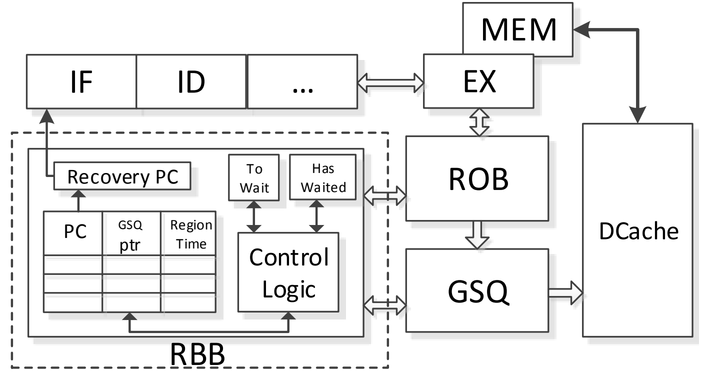

Fig. 4. The high-level view of Turnstile hardware scheme

the threshold, if one exists, and splits the bb_i (line 14) into head ($bb_i{}'$) and tail (bb_i) basic blocks. If there is no such store, Turnstile puts the new boundary at the basic block entry. Then, Turnstile updates the accumulating store number (accum_str) with the total store number Str_{bb_i} of the new (tail) basic block bb_i (line 17). The above process keeps repeated as long as accum_str is greater than the threshold. Finally, for each successor basic bb_j of bb_i, Turnstile updates the incoming store number of bb_j ($IncomeStr_{bb_j}$) with the remaining accum_str. $IncomeStr_{bb_j}$ is updated iff the remaining accum_str is greater than $IncomeStr_{bb_j}$'s original value (line 19$\sim$21). Note that, as Turnstile traverses the CFG in program topological order, the incoming store number of each basic block is always guaranteed to be the maximum incoming store number from all its predecessor basic blocks when Turnstile visits the basic block.

Example: Figure 3 presents an example of Turnstile's region formation heuristic. Figure 3 (a) shows the original CFG where the number in each basic block is the total store number. Assuming the basic block BB_0 starts with a region boundary Rg_0 (shown as a dotted line), Turnstile traverses the basic blocks in a topological order ($BB_0 \rightarrow BB_1 \rightarrow BB_2 \rightarrow BB_3 \rightarrow BB_4$). We also assume a store queue with 40 entries, thus the threshold is 20 in this case. When Turnstile visits BB_3, as the calculated accumulating store number (accum_str) is already 50 thus greater the threshold.

Turnstile then keeps placing a region boundary thus splitting BB_3 until the accum_str becomes smaller than the threshold. For ease of presentation, we assume that the sum of each recalculated total store number for every split blocks remain the same (45) as shown in Figure 3 (b). The figure shows the partitioned program where the numbers on the edges are the incoming store numbers. Note that the maximum incoming store number of BB_4 (i.e., $IncomeStr_{BB_4}$) is 20. Thus, we place a boundary at the entry of BB_4 as its accum_str is greater than the threshold.

C. Optimization for Storeless Loops

In order to improve the performance, Turnstile identifies those loops that have no store in the loop body, i.e., storeless loops. Here, Turnstile can eliminate the region boundaries in the header of such loops, since there is no chance of the store queue overflow during the execution of the loops. As a result, Turnstile can avoid inserting checkpoints (i.e., store) right after the instructions that update the register region inputs in the loop. However, the updated registers in the loop may be able to reach the later regions once the loop terminates.

To preserve the live-out registers, Turnstile places a region boundary at the end of loop preheader, and creates new basic blocks before the loop's exit basic blocks. Note that the new blocks exist outside the loop. Then, Turnstile inserts checkpoints into those new basic blocks, and places a region boundary at the end of the new blocks. With this optimization, Turnstile still guarantees 100% error recovery while significantly improving the performance.

V. Turnstile Hardware Support: Region Boundary Buffer (RBB)

Turnstile leverages special hardware logic to realize the region verification and the recovery for soft error resilience. As highlighted with the dashed box in Figure 4, the hardware logic interacts with the ROB and the gated store queue (GSQ) to recognize the boundary of executed regions and to write back their verified data, respectively. Note that Turnstile honors the original semantics of the store queue in an Out-of-Order processor (e.g. CAM search, Non-snoopable *etc.*).

To keep track of those regions that have not been verified yet, the hardware logic leverages the region boundary buffer (RBB) whose entry is allocated whenever the boundary instruction is executed. The RBB entry is a tuple of three values: (1) the PC of the boundary instruction; once the region ending at the boundary is verified, the PC is used as a recovery point when an error is detected, (2) the tail pointer of GSQ; this is necessary to write back only the data written in the fault-free region once it is verified, (3) the execution cycles of the region; this is important information for Turnstile's efficient region verification in the next section. Whenever a new region is verified, the logic writes the PC of the boundary instruction that finishes the region to a special register called RP (recovery PC) which will be used in case of an error.

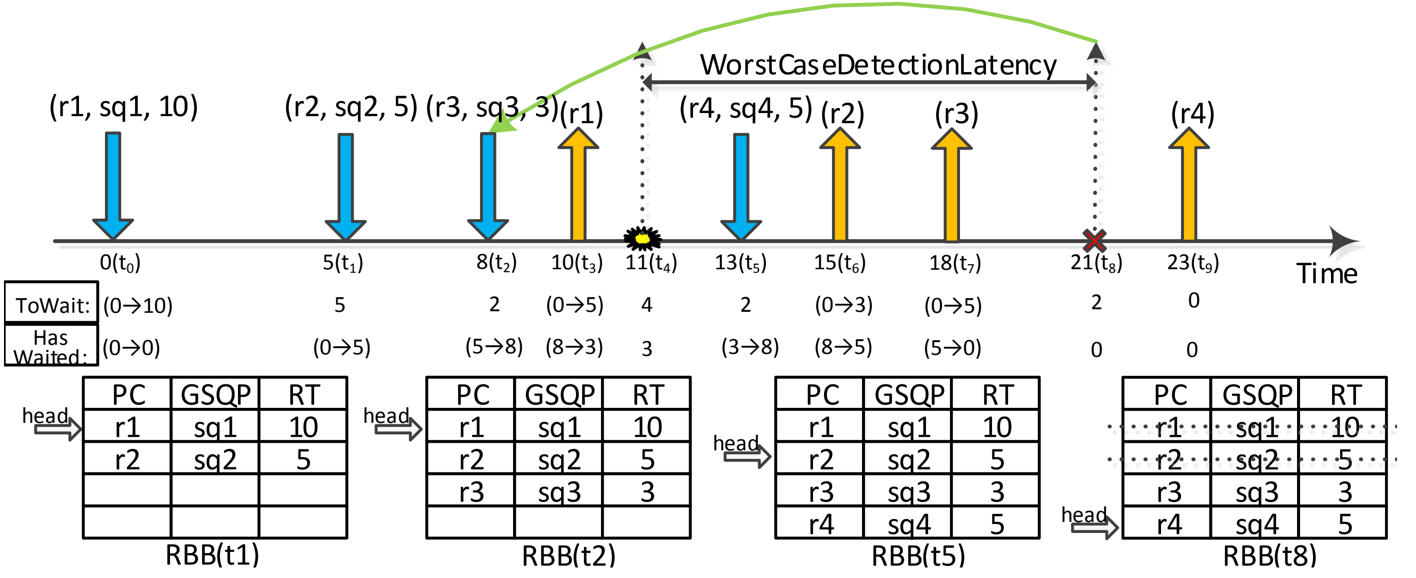

Fig. 5. An illustrating example of Turnstile's hardware support.

A. Region Verification and Recovery Algorithm

This section first introduces a simple but an inefficient region verification/recovery algorithm and the lessons from it. Then, Turnstile's efficient algorithm is presented with an example.

Naive Algorithm: On a region boundary, one would write the TSC (time stamp counter) in the tail index of RBB, checking it from the head to see if there is an entry whose TSC value is less than the difference between the current TSC and the error detection latency time (cycles). Obviously, this naive approach is expensive due to the access to multiple entries of RBB on each region boundary, the check cost, and the area overhead (64-bit-TSC) in RBB entries. Even worse, for the region that has already spent the detection latency cycles since it is finished, the GSQ cannot release its verified data until the next region boundary is reached. Note that this makes GSQ become full more frequently than usual, thus causing pipeline stalls leading to performance degradation.

Turnstile's Efficient Algorithm: The lesson from the previous algorithm is three-fold. First, TSC based approach is expensive, e.g., 64 bits per each RBB entry; Turnstile only uses $Log_2 DL$ bits where DL is the error detection latency in cycles. Second, the verification check should be done in a real time manner, i.e., it should not be delayed to the next region boundary. Third, as soon as a region is verified, Turnstile should be able to write back all its data buffered in GSQ without unnecessarily holding them. In light of this, Turnstile leverages a timer based approach to efficiently achieve region verification based on the following insight.

Axiom 1: If a given region, R_n, is verified at a time T, then the very next region, R_{n+1}, will be verified at a later time T', $T' = T + ElapsedTime(R_{n+1})$ where $ElapsedTime$ maps a region to its execution cycles.

The takeaway is that Turnstile can achieve the region verification by simply tracking the execution cycles of each executed region. For this purpose, Turnstile requires only two metadata for tracking every region. (1) a **watchdog timer** called $ToWait$ represents the number of cycles for which the region in the head of RBB should wait more before being verified. (2) an auxiliary **counter** called $HasWaited$ represents the number of cycles for which a region has already waited to be verified once the region becomes the head of RBB. Thus, $ToWait$ and $HasWaited$ both track the head entry of RBB.

When a region boundary instruction is executed, Turnstile allocates a new RBB entry for the region that has just finished at the boundary. At the same time, Turnstile reads the watchdog timer ($ToWait$) and the counter ($HasWaited$) to calculate the elapsed time of the region, i.e., $RegionTime$. Note that unlike the watchdog timer that automatically counts down each cycle, Turnstile here reads the value of $HasWaited$ which was written at the previous region boundary. Therefore, the $RegionTime$ can be obtained by subtracting the sum of $HasWaited$ and $ToWait$ from the error detection latency (DL). Then, Turnstile records the resulting $RegionTime$ in the new RBB entry allocated, and updates the $HasWaited$ counter as "$DL - ToWait$".

When $ToWait$ becomes zero, i.e., RBB's head entry region is now verfied, it is removed by updating the head pointer of RBB; the removed entry is referred to as the old head entry hereafter. Thus, the PC of the old head entry is written in RP that will be used for recovery in case of an error. Then, GSQ marks as verified those entries preceding the GSQ pointer of the old head entry, so that they can be written back to L1 data cache when the bandwidth between the store queue and the cache is available. Note, having the old head entry tracked by $ToWait$ and $HasWaited$ removed from RBB requires updating them for a new head entry region. $ToWait$ is reset

to the $RegionTime$ of the new head entry of RBB (See the Axiom 1), and the value of $RegionTime$ is subtracted from $HasWaited$. Recall that $HasWaited$ means the time for which a region has waited to verify itself since it became the head entry of RBB. Therefore, excluded is the time between the end of the old head region and the end of the new head region, which is the execution cycles of the new head region, i.e., its $RegionTime$.

If an error is detected, Turnstile's recovery handler discards all unverified data in the GSQ and empties the RBB. Then, Turnstile restores the checkpointed register values from memory. At this point, all the program status is guaranteed to be the same as it were before the error occurs. Finally, Turnstile jumps back to the end of the most recently verified region whose address is available in RP, the recovery point register.

Summary of Hardware Cost: Turnstile's hardware support is lightweight and non-intrusive to the critical path. It only needs a 5 bit timer and 5 bit counter. The modification to store queue for gating the stores requires one bit to flag verification status for each entry. The RBB is also trivially small (<14 entries) as evaluated in Section VII-E.

Example: Figure 5 presents an example to show how RBB verification works. We assume the worst-case error detection latency (WCDL) to be 10 cycles. Along the timeline, the blue arrow corresponds to the time points on which a region boundary instruction is executed, while the orange arrow to the time points on which a region is verified.

When a region boundary instruction is executed, i.e., a blue arrow is reached in the timeline, the RBB is updated with the tuple above the arrow comprising the PC of the boundary instruction, the tail of pointer the GSQ, and RegionTime shown as RT in Figure 5. Below the timeline, there are two lines of numbers which represent the $ToWait$ timer and $HasWaited$ counter at each time point, respectively. When the $ToWait$ timer expires reaching 0, RBB first pops out the head entry, and updates the timer with the RegionTime in the new head entry. At the same time, RBB also updates the $HasWaited$ accordingly.

Below the two lines of numbers, there are four tables showing the status of RBB at different time points, i.e., t1, t2, t5, and t8. At t1, there are two region entries sitting in the table waiting to be verified. At t2, both the $ToWait$ timer and $HasWaited$ counter are not zero in the RBB where the RT is calculated by subtracting the sum of $HasWaited$ and $ToWait$ from the WCDL. At t3, the region r1 has already been verified. Therefore, the head pointer of RBB comes to point to r2 at t3, and the $ToWait$ timer is updated with r2's RT value. At the same time, r2's RT value is subtracted from $HasWaited$ to update the $HasWaited$ counter.

At t4, a soft error occurs, and it is detected at t8 after the WCDL. At t8, r3 has already been verified since t7 and popped out from the RBB, which means the instructions before r3 has been verified. Therefore, at t8, Turnstile redirects program control to the end of the most recently verified region (r3 in this case), thus re-executing r4 from the beginning to recover from the error. The takeaway is that Turnstile can achieve the region verification by simply tracking the execution cycles of each executed region.

VI. Discussion and Limitations

Turnstile Hardware Protection/Fault Model: We assume that both the gated store queue (GSQ) and the region boundary buffer (RBB) are protected against soft errors. Note that there will not be any timing delay as they are off the critical path. Besides, we also assume that our watchdog timer and counter are also hardened by some protection schemes.

Tolerating Multiple-bit Errors: Multiple-bit errors can be handled easily by Turnstile. As discussed in Section III, all the errors are guaranteed to be detected by the sensor-based detection scheme, and any un-verified stores are discarded during recovery.

Handling I/O Operations: Turnstile takes a conventional way, i.e., holding the I/O stores during the detection-latency cycles in an ECC-protected buffer to ensure error-free I/O stores as well as replaying the I/O loads upon recovery as with [3], [12]. Turnstile can have a gated I/O buffer, which is similar to our gated store queue (GSQ), to verify I/O requests in case they bypass the GSQ.

Silent Data Corruption (SDC): Even if an energetic particle strike is the major source of soft errors, they can also be induced by other sources, e.g., transistor variability or power supply noise *etc.* [36], [37]. Since these sources are not covered by the sensor-based soft error detection, Turnstile might generate silent data corruption (SDC). However, under aggressive transistor scaling and near-threshold computing, high-energy particle strikes are considered a critical source of soft errors [2], [38], [39].

VII. Evaluation and Analysis

Our evaluation answers the following research questions:

- What is the performance impact of our hardware support and how sensitive is it to the worst-case detection latency (WCDL)?
- How does our compiler transformation affect the application performance and region characteristics?
- What is the impact on overall performance when Turnstile functions as a whole? How is the overhead affected by varying the size of the gated store queue (GSQ) and the time of the WCDL?
- What is the right size of the region boundary buffer (RBB)?

A. Experimental Methodology

We implemented the compiler analysis and checkpoint instrumentation passes described in Section IV using the LLVM Compiler Infrastructure [40]. As with the common practice in the literature [41], [10], We used SPEC2006 [42], MediaBench [43] and SPLASH2 [44] benchmark suites targeting different computing areas for our experiments, and all applications were compiled with standard -O3 optimization.

We conduct our simulations on the Gem5 simulator [45] with the ARMv7 ISA, modeling a modern two-issue out-of-order 2 GHz (1∼4 cores) processor with private L1-I/D

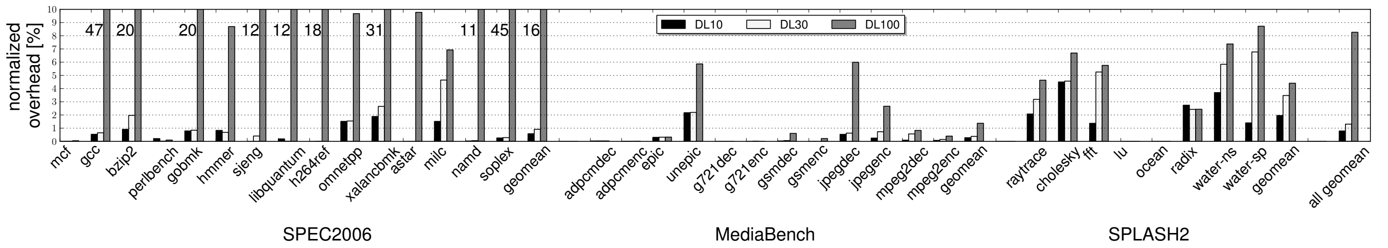

Fig. 6. Turnstile hardware effect varying different detection latencies (10, 30, 100)

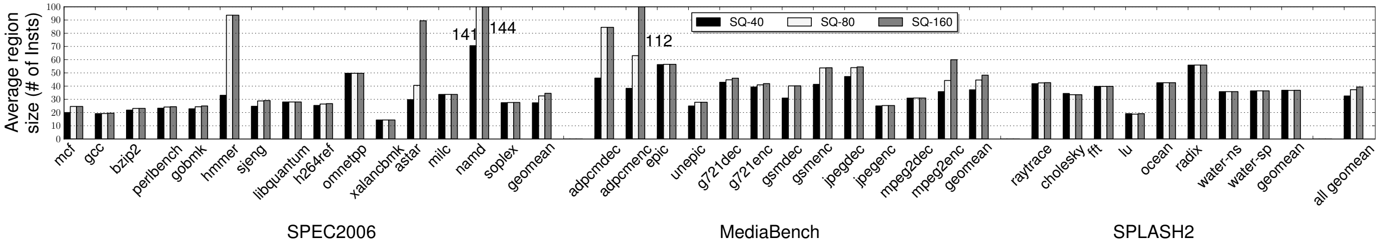

Fig. 7. The average region length when partitioned for different store queue size (40, 80, 160)

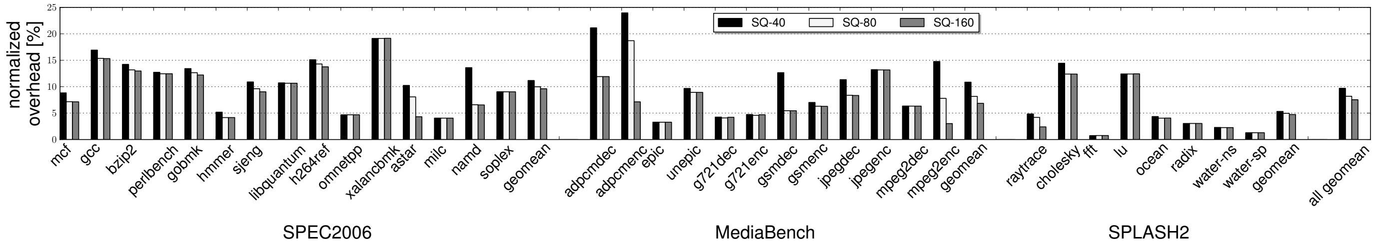

Fig. 8. The normalized instruction counts when partitioned for different store queue size (40, 80, 160)

(32KB, 2-way, 2-cycle latency, LRU), and shared L2 (2MB, 8-way, 20-cycle latency, LRU) caches. The ROB, physical integer register file, and load queue have 192, 256, and 40 entries, respectively. The simulator was modified to accurately implement the effect of the gated store queue (GSQ) and the region boundary buffer (RBB) as discussed in Section V. We vary the size of the GSQ (i.e., 40, 80, and 160 entries) to perform the sensitivity analysis.

For SPEC2006 applications, in order to show the performance impact of the executables generated by Turnstile's compiler versus the original executables, we synchronize the simulation length by measuring the number of functions executed which is constant between two versions as in the case of the prior work [19]. All the benchmarks in SPEC2006 are fast-forwarded the number of function calls to execute at least 5 billion instructions on the original executables, and then we simulate 1 billion additional instructions on the original executables. For the MediaBench and SPLASH2, we simulate the entire program for all the applications.

B. Turnstile Hardware Effect

Figure 6 examines the effect of Turnstile's hardware support by varying the WCDL. We partitioned the original binaries into different regions. However, we do not inject the checkpointing stores. We use the partitioned binaries as input and run the experiments on our modified simulator to observe how Turnstile's hardware support affects the performance of original binaries. We normalize the overhead with the baseline where the original binaries run on an un-modified simulator.

From left to right, each application has three bars representing different worst-case error detection latencies (i.e., 10, 30, 100) which are the time needed to verify the region. According to Upasani's work [3], having a 30 or 100 WCDL cycles of sensor-based detection scheme will cost less than 1% area overhead. However, as we assume that the memory system is protected by other schemes, e.g., ECC, there is no need to protect the caches, etc. That is, it is possible to deploy those detectors on only the core parts to achieve a lower WCDL, thus we also explore the effect of 10 WCDL cycles.

It is interesting to observe that the applications in three different benchmark suites show significantly different degrees of sensitivity to different WCDL. As the applications in MediaBench are mostly computation-intensive, they are rarely affected by Turnstile's hardware even when the WCDL is 100 cycles. On the other hand, the applications in SPLASH-2 benchmark show relatively higher overhead, and are more sensitive to WCDL than the others. The reason is that Turnstile treats fence operations and atomic operations, which are the basic ingredients in implementing synchronization between threads, as region boundaries. This implies that the cost of such

operations become more expensive than before as Turnstile hardware have to wait for WCDL, affecting other threads. However, our experiment shows that the overall overhead is about 3.5% for WCDL of 30 and 4.5% for WCDL of 100.

Not surprisingly, with longer WCDL, the resulting performance overhead is higher. With 100-cycles-of-WCDL, Turnstile can slow down the original program up to 47% with 16% performance degradation on average. This is because holding the stores for a long time may result in the pipeline stalls when the gated store queue (GSQ) become full. As expected, smaller WCDL of 10 does little harm to the program performance which is less than 1% degradation in average. It is interesting to find that 30-cycles-of-WCDL also have a trivial impact on the performance (around 1%). To sum up, Turnstile can spend less than 1% hardware overhead to achieve 30-cycles-of-WCDL detection scheme, and the runtime overhead of Turnstile's hardware is not significant. Thus, in order to achieve low-cost, the following sections evaluate Turnstile with 10, and 30 cycles of WCDL configuration. In addition, we also evaluate Turnstile with 5 cycles WCDL assuming an aggressive sensor placement to reduce the WCDL [3].

C. Turnstile Compiler Effect

This section shows how Turnstile's compiler affects the original binaries. We report 1) the average number of instructions per region, and 2) the number of increased instructions for checkpointing, when different region formation thresholds are used.

1) Region Length: Figure 7 shows the average dynamic region length by varying the gated store queue (GSQ) size (i.e., 40, 80, 160) during the region formation. Note that we use the half GSQ size as the region formation threshold. However, for most of the applications, increasing the threshold does not help improve the dynamic region length. This is because we have to place the region boundary at the beginning of the header for most loops that have stores in the loop body to avoid the GSQ overflow. As such loops dominate the execution time, we can expect that the dynamic region length should not vary a lot even with different thresholds during the region formation.

A few applications (e.g., hmmer, astar and adpcm etc.) show great improvement in dynamic region length after increasing the threshold. This is because these applications have few loops with store instruction in the loop body or have a large loop body where increasing the threshold improves the dynamic region length.

As Turnstile does not require the error to be detected within the each region. The region length actually has no impact on the error coverage. In fact, a longer region may be less preferable during recovery as we may waste a lot of execution time. However, shorter regions are not always good for Turnstile, as Turnstile need to preserve the register region inputs with checkpointing store instruction. The shorter the region length is, the more performance overhead we need to pay for the checkpointing. Therefore, mediocre region length may be more preferable to Turnstile. The average dynamic region length is around 35 instructions across different GSQ sizes which is beneficial for both tolerating false-positives and performance.

2) Checkpointing Instructions: Figure 8 shows the number of increased instructions for checkpointing normalized to the original binary. The average instruction overhead of Turnstile is around 10% when regions are formed for the gated store queue (GSQ) of size 40. The general trend is that when increasing the GSQ sizes (region formation threshold) which in turn generate less regions, the checkpoint instruction overhead gets smaller as fewer region inputs need to checkpoint.

D. Overall Overhead

This section evaluates the overall overhead by running our Turnstile compiler generated binaries on top of Turnstile's hardware support. We vary the factors (e.g. size of GSQ and WCDL) to see how it can affect the overall performance. Our baseline is the original binaries that run on an un-modified simulator.

1) Sensitivity to WCDL: Figure 9 show the the normalized runtime overhead with the gated store queue (GSQ) of size 40 and different WCDL from 5, 10, to 30, compare to the baseline. Increasing the WCDL can slow down some applications' performance by up to 20%, but the geometric mean (6$\sim$8%) shows trivial differences between different WCDL.

2) Sensitivity to the Size of Gated Store Queue: Figure 10 show the normalized runtime overhead compared to the baseline with varying the size of the gated store queue (GSQ). The overall overhead across different GSQ sizes stay around 6% up to 7.2%. With a large GSQ, Turnstile compilers can create a longer region, leading to less checkpoint instructions (Section VII-C2). The general trend observed here is that the increase of GSQ's size helps applications reduce the runtime.

E. Exploration of Region Boundary Buffer Size

Lastly, we explore the design choice of the size of the region boundary buffer (RBB). Figure 11 shows the dynamic number of entries occupied during the execution. We use a gated store queue (GSQ) with 40 entries and a WCDL of 30 cycles to perform our profiling simulation. The left bar shows the dynamic average number of entries used by Turnstile while the right bar shows the maximum number of RBB entries occupied. As we can see, the maximum number across all the applications is 14 while the average number is 2. The number of bits needed in an entry of RBB can be calculated as follows:

$$\#RBB_Entry = \#(PC) + Log_2\#(GSQ) + Log_2\#(WCDL)$$

where #(PC) represents instruction size (32), #(GSQ) is the number of entries in GSQ (40) and #(WCDL) is the cycle number of WCDL (30).

Therefore, one entry in the RBB requires 43 bits. Even if we design the RBB with the maximum number (i.e. 14). We only need 602 bits for the RBB. Besides, Turnstile also requires a 5 bit timer and 5 bit counter. The overall hardware overhead is trivial. More importantly, Turnstile's hardware implementation is off the critical path of the processor.

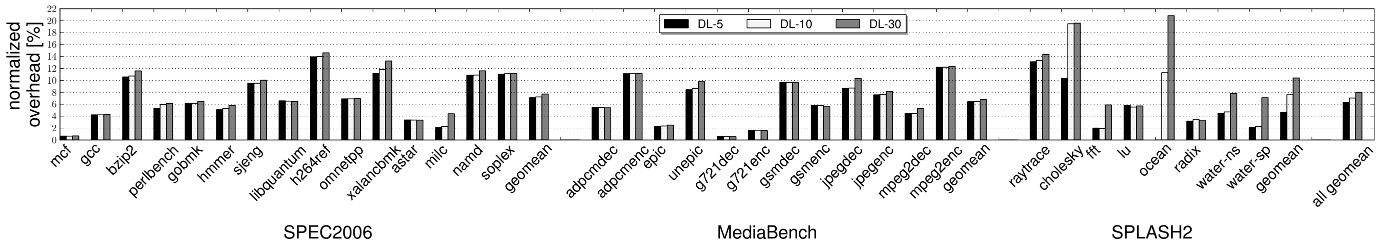

Fig. 9. Turnstile overhead with 40 gated store queue entries varying different detection latencies (5, 10, 30)

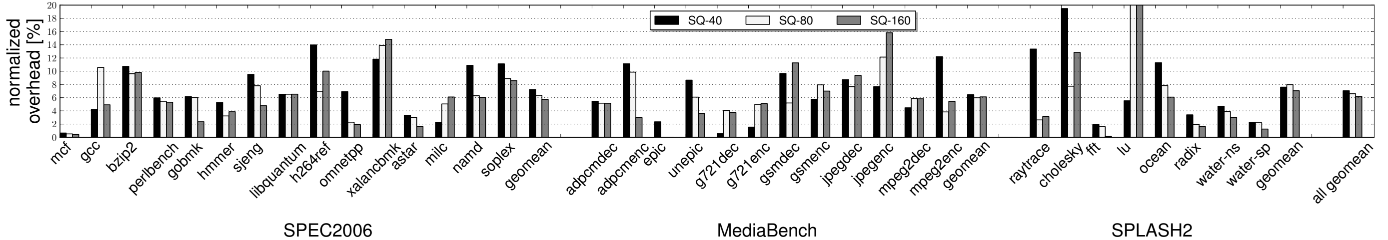

Fig. 10. Turnstile overhead with 10 cycles WCDL varying different gated store queue size (40, 80, 160)

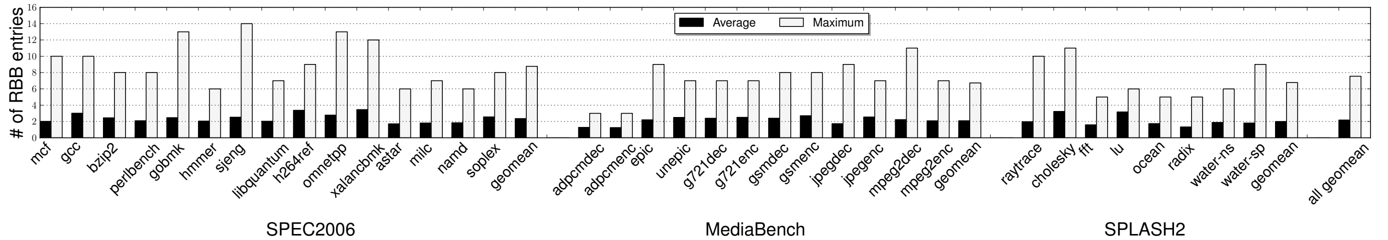

Fig. 11. Region boundary buffer dynamic entry number with 40 gated store queue entries and 30 cycles WCDL

VIII. Other Related Work

In addition to the recovery schemes discussed previously, this section compares Turnstile with other existing recovery schemes and shows how Turnstile differs from those recovery schemes.

Coarse-Grained Recovery: Prior coarse-grained recovery schemes [3], [14] generally cannot contain the error within the core and requires expensive hardware support to relieve the performance overhead which relegates their use to high-end server systems. For example, to preserve the register states, those recovery schemes usually leverage two additional copies of ECC-protected register file (RF) and register alias table (RAT). As for the memory state preservation, previous coarse-grained recovery schemes either restructure the caches (e.g., new coherence) to incrementally checkpoint the memory state [3] or maintain a large buffer for memory logging [14]. Besides, Jeyapaul et al. [46] explores multicore CMP architecture to recover from soft errors with an efficiently modified cache structure. In contrast, Turnstile does not require such expensive hardware support and provides a low-cost alternative.

Finer-Grained Recovery: There are several recovery schemes that can also contain the error within the core. Flushing the pipeline to recover from a soft error is another alternative [47]. However, such schemes requires a near-zero detection latency detector and cannot be applied to low-cost acoustic sensor based detector with detection latency. Triple-Modular-Redundancy [48] can correct the error at the instruction-level at the cost of significant performance degradation. On the contrary, Turnstile leverages the low-cost acoustic sensors and imposes low performance overhead making it a practical choice for soft error resilience.

IX. Summary

This paper introduces Turnstile, a lightweight hardware/software cooperative soft error resilience technique that leverages low-cost acoustic sensors. Turnstile can provide guaranteed soft-error recovery that can contain the error within the core, incurring only ∼8% runtime overhead without expensive hardware support for checkpointing. Leveraging Turnstile's novel compiler analysis and lightweight hardware support, we can preserve the program state effectively and practically. Turnstile neither requires expensive RF protection to preserve the register state nor a extra large store buffer to maintain the memory state. To the best of our knowledge, Turnstile is the first technique for soft error resilience, that does not impose expensive hardware overhead, with guaranteed recovery. We believe that Turnstile can lay the foundation for the soft error resilience of future computing systems.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments and feedback. This work was

partly supported by the National Science Foundation under the grant CCF-1527463 and Google Faculty Research Awards. This work was also suppported by and used the resources of Oak Ridge National Laboratory, which is managed by UT Battelle, LLC for the U.S. DOE (under the contract No. DE-AC05-00OR22725).

REFERENCES

[1] L. Wang *et al.*, "Implications of the power wall: Dim cores and reconfigurable logic," *IEEE Micro*, pp. 40–48, 2013.

[2] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," *Micro, IEEE*, vol. 23, pp. 14–19, July 2003.

[3] G. Upasani, *et al.*, "Avoiding core's due & sdc via acoustic wave detectors and tailored error containment and recovery.," in *ISCA*, pp. 37–48, 2014.

[4] H. Esmaeilzadeh, *et al.*, "Dark silicon and the end of multicore scaling," in *ISCA*, pp. 365–376, 2011.

[5] N. Hardavellas, *et al.*, "Toward dark silicon in servers," *IEEE Micro*, vol. 31, no. 4, pp. 6–15, 2011.

[6] M. Shafique, *et al.*, "The eda challenges in the dark silicon era: Temperature, reliability, and variability perspectives," in *DAC*, pp. 185:1–185:6, 2014.

[7] H. Kaul, *et al.*, "Near-threshold voltage (ntv) design: Opportunities and challenges," in *DAC*, pp. 1153–1158, 2012.

[8] G. Upasani, *et al.*, "Setting an error detection infrastructure with low cost acoustic wave detectors.," in *ISCA*, pp. 333–343, 2012.

[9] G. Upasani, *et al.*, "A case for acoustic wave detectors for soft-errors," *IEEE Transactions on Computers*, vol. 65, pp. 5–18, Jan 2016.

[10] G. A. Reis, *et al.*, "Swift: software implemented fault tolerance," in *CGO*, pp. 243–254, March 2005.

[11] N. J. Wang *et al.*, "Restore: symptom based soft error detection in microprocessors," in *DSN*, pp. 30–39, June 2005.

[12] E. Rotenberg, "Ar-smt: a microarchitectural approach to fault tolerance in microprocessors," in *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pp. 84–91, June 1999.

[13] M. Dimitrov *et al.*, "Unified architectural support for soft-error protection or software bug detection," in *PACT*, pp. 73–82, 2007.

[14] D. Sorin, *et al.*, "Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *ISCA*, pp. 123–134, 2002.

[15] A. Meixner, *et al.*, "Argus: Low-cost, comprehensive error detection in simple cores," in *MICRO*, pp. 210–222, IEEE, 2007.

[16] S. S. Mukherjee, *et al.*, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, (Washington, DC, USA), pp. 29–, IEEE Computer Society, 2003.

[17] J. Suh, *et al.*, "Soft error benchmarking of l2 caches with parma," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, pp. 85–96, June 2011.

[18] S. Feng, *et al.*, "Encore: low-cost, fine-grained transient fault recovery," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 398–409, ACM, 2011.

[19] M. A. de Kruijf, *et al.*, "Static analysis and compiler design for idempotent processing," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, (New York, NY, USA), pp. 475–486, ACM, 2012.

[20] M. de Kruijf *et al.*, "Idempotent code generation: Implementation, analysis, and evaluation," in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2013.

[21] Q. Liu, *et al.*, "Clover: Compiler directed lightweight soft error resilience," in *LCTES*, (New York, NY, USA), pp. 2:1–2:10, ACM, 2015.

[22] X. Li *et al.*, "Application-level correctness and its impact on fault tolerance," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 181–192, IEEE, 2007.

[23] M. J. Wing *et al.*, "Gated store buffer for an advanced microprocessor," *U.S. Patent 6,011,908 A*, January 2000.

[24] M. S. Gupta, *et al.*, "Decor: A delayed commit and rollback mechanism for handling inductive noise in processors," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pp. 381–392, Feb 2008.

[25] Q. Liu *et al.*, "Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems," in *Proceedings of the IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2016.

[26] S. Muchnick, *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[27] B. S. Gill, *et al.*, "An efficient bics design for seus detection and correction in semiconductor memories.," in *date05*, pp. 592–597, IEEE Computer Society, 2005.

[28] Z. F. Huang *et al.*, "BISS: A Built-In SEU Sensor for Soft Error Mitigation," *Applied Mechanics and Materials*, vol. 130, pp. 4228–4231, Oct. 2011.

[29] A. Narsale *et al.*, "Variation-tolerant hierarchical voltage monitoring circuit for soft error detection.," in *isqed09*, pp. 799–805, IEEE, 2009.

[30] S.-L. Gong, *et al.*, "Clean-ecc: High reliability ecc for adaptive granularity memory system," in *the Proceedings of MICRO*, 2015.

[31] J. Kim, *et al.*, "Frugal ecc: Efficient and versatile memory error protection through fine-grained compression," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, (New York, NY, USA), pp. 12:1–12:12, ACM, 2015.

[32] ARM., "Cortex-a57 technique reference manual."

[33] Q. Liu, *et al.*, "Compiler directed soft error detection and recovery to avoid due and sdc via tail-dmr," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. XX, no. X, 2016.

[34] Q. Liu, *et al.*, "Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2016.

[35] C. Blundell, *et al.*, "Invisifence: Performance-transparent memory ordering in conventional multiprocessors," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 233–244, 2009.

[36] W. Jang, *Soft-error tolerant quasi delay-insensitive circuits*. PhD thesis, Pasadena, CA, USA, 2011.

[37] S. S. Mukherjee, *et al.*, "The soft error problem: An architectural perspective," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pp. 243–247, 2005.

[38] N. DeBardeleben, *et al.*, "Extra bits on sram and dram errorsmore data from the field," in *IEEE Workshop on Silicon Errors in Logic-System Effects (SELSE)*, 2014.

[39] J. Wadden, *et al.*, "Real-world design and evaluation of compiler-managed gpu redundant multithreading," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 73–84, IEEE Press, 2014.

[40] C. Lattner *et al.*, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.

[41] S. Feng, *et al.*, "Shoestring: Probabilistic soft error reliability on the cheap," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 385–396, ACM, 2010.

[42] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[43] C. Lee, *et al.*, "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, (Washington, DC, USA), pp. 330–335, IEEE Computer Society, 1997.

[44] S. Woo, *et al.*, "The splash-2 programs: characterization and methodological considerations," in *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pp. 24–36, June 1995.

[45] N. Binkert, *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, Aug. 2011.

[46] R. Jeyapaul, *et al.*, "Unsync-cmp: Multicore cmp architecture for energy-efficient soft-error reliability," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 1, pp. 254–263, 2014.

[47] P. Racunas, *et al.*, "Perturbation-based fault screening," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 169–180, IEEE, 2007.

[48] J. Chang, *et al.*, "Automatic instruction-level software-only recovery," in *International Conference on Dependable Systems and Networks (DSN'06)*, pp. 83–92, June 2006.