2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 978-1-6654-6272-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/MICRO56248.2022.00030

Featherweight Soft Error Resilience for GPUs

Yida Zhang
Purdue University
zhan3339@purdue.edu

Changhee Jung
Purdue University
chjung@purdue.edu

***Abstract*—This paper presents Flame, a hardware/software co-designed resilience scheme for protecting GPUs against soft errors. For low-cost yet high-performance resilience, Flame uses acoustic sensors and idempotent processing for error detection and recovery, respectively. That is, Flame seeks to correct any sensor-detected errors by re-executing the idempotent region where they occurred. To achieve this, it is essential for each idempotent region to ensure the absence of errors before moving on to the next region. This is so-called soft error verification that takes sensors' worst-case detection latency (WCDL) to verify each region finished. Rather than waiting for WCDL at each region end, which incurs too much performance overhead, Flame proposes WCDL-aware warp scheduling that can hide the error verification delay (i.e., WCDL) with GPU's inherent massive warp-level parallelism. When a warp hits each idempotent region boundary, Flame deschedules the warp and switches to one of the other ready warps—as if the region boundary were a regular long-latency operation triggering the warp switching. By leveraging GPU's inherent ability for the latency hiding, Flame can completely eliminate the verification delay without significant hardware modification. The experimental results demonstrate that the performance overhead of Flame is near zero, i.e., 0.6% on average for 34 GPU benchmark applications.**

I. Introduction

Resilience against soft errors is as important as performance and energy efficiency for any computer system due to their direct impact on correctness [1], [2], [3]. One of the major sources of soft errors is the striking of energetic particles, e.g., cosmic rays and alpha particles, on the circuit. The resulting bit flips may lead to program crashes or silent data corruption (SDC) where the errors are not detected during program execution but end up with wrong output at program termination [4], [5], [6], [7], [8]. Unfortunately, with technology scaling, electronic circuits are becoming more vulnerable to such radiation-induced soft errors[1] [5], [8], [9], [10], [11].

Given that GPUs are used everywhere from embedded systems (e.g., drones and self-driving cars) to HPC systems (e.g., data centers and supercomputers) [12], [13], [14], [15], it is now more critical than ever to protect GPUs against soft errors without sacrificing GPU's benefits of high performance and energy efficiency. Hence, starting from Fermi, Nvidia GPUs are equipped with error correction code (ECC) to protect their memory hierarchy, i.e., a register file (RF), caches, DRAM [16]. Nevertheless, since ECC protects only the data array, the entire GPU pipelines remain vulnerable to soft errors.

One easy way to detect soft errors during the pipeline execution is running each instruction twice and comparing the outputs of the original instruction and its replica. However, when such instruction duplication is naively performed for GPUs, the resulting performance overhead is too significant ($\approx$ 50% slowdown) [1], [17]—though it outperforms compiler-managed GPU redundant multithreading [12]. This motivates researchers to focus on optimizing instruction duplication based error detection. For example, Mahmoud *et al* propose GPU-specific optimizations, that trade off fine-grained recoverability for performance, reducing the error detection overhead to 36% on average [1]. Later on, SwapCodes [17] proposes to pair the original instruction output with the ECC code of the replica instruction output. That way if the outputs mismatch, it can be detected by ECC's error checking logic without explicitly comparing the outputs. While SwapCodes further reduces the average overhead to 15-21%, it is still costly on a large scale as in data centers and supercomputers. Thus, there is a compelling demand for lightweight GPU soft error resilience.

Interestingly, CPU architects have found a way to achieve SDC-free soft error resilience on the cheap, using acoustic sensors that detect errors with sensing the sound wave caused by particle strikes [8], [18], [19], [20], [21], [22]. Since they always result in such sound wave [23], any radiation-induced soft error can be detected within the worst-case detection latency (WCDL)[2] of the sensors. The upshot is that they enable so-called *soft error verification*; program execution till any time point will be verified to be error-free WCDL cycles later after the point, provided no error is detected during the WCDL cycles—that is referred to as verification delay. Moreover, the sensors are cost-effective, e.g., 30 cycles of WCDL for a 2GHz out-of-order CPU can be made with only 300 sensors causing less than 1% area overhead [19], [20], [21], [22].

With that in mind, this paper presents Flame, the first lightweight soft error resilience scheme for GPUs

[1]For simplicity, we refer to the radiation-induced soft errors, i.e., those caused by mainly cosmic rays and alpha particles, as soft errors hereafter.

[2]WCDL is determined at design time by the number of sensors deployed and their network topology.

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00030

that achieves near-zero performance overhead. Unlike duplication-based prior resilience studies [1], [12], [17], Flame leverages acoustic sensors to detect soft errors at a low cost. For recovery, Flame partitions the program into a series of idempotent regions that can be re-executed multiple times and still result in the same correct output—as long as the input is preserved [24]. That is, Flame seeks to correct any sensor-detected errors by re-executing the idempotent region where they occurred. To achieve this, it is essential for each idempotent region to verify the absence of errors at the end of the region by waiting for WCDL before moving on to the next region; otherwise, errors may escape to the next region and corrupt its inputs, thereby making it non-idempotent. By verifying all regions one by one, Flame can achieve correct idempotent recovery; if an error is detected, Flame can roll back to the most recently verified idempotent region boundary and resume from the region boundary to recover from the error.

Unfortunately, the soft error verification delay (i.e., WCDL) between the regions—simply region verification delay— incurs a significant performance overhead on GPUs. To address the problem, Flame proposes WCDL-aware warp[3] scheduling that can hide the region verification delay with GPU's inherent massive warp-level parallelism. When a warp hits each idempotent region boundary, Flame deschedules the warp and switches to one of the other ready warps—as if the boundary were a regular long-latency operation triggering the warp switching. By leveraging the GPU's inherent latency hiding capability, Flame can completely eliminate the region verification delay without significant hardware modification.

However, this presents a new challenge for soft error recovery. Although an error is detected during the execution of an active warp, the error might be caused by some other warp that has already been descheduled—for verification—by the WCDL-aware warp scheduling, in which case re-executing the active warp cannot recover from the error. To ensure correct error recovery for such a problematic case, Flame devises simple hardware support that re-executes all unverified warps, whether or not they are active, upon error detection by resetting the PCs to their most recent region boundaries respectively. In this way, Flame can correctly recover from soft errors even if the active warp when they are detected is not the one where they occurred.

The experimental results demonstrate that the performance overhead of Flame is near zero, i.e., 0.6% on average for 34 GPU benchmark applications. Our contributions can be summarized as follows:

- Flame is the first lightweight GPU resilience scheme that can protect the entire GPU pipeline—based on acoustic sensor based error detection and idempotent processing based error recovery.
- Flame identifies a certain pattern in GPU program that degrades performance with the acoustic sensor based error detection and presents an optimization technique to resolve the performance problem.
- Flame incurs a near-zero performance overhead while its hardware complexity is minimal, on top of the area overhead ($< 1\%$) of the acoustic sensors deployed.

[3]Group of threads in Nvidia's GPU.

II. Background

A. Acoustic Sensor Based Soft Error Detection

Soft errors, also known as transient faults, are often the cause of failures in today's computing systems. Given that a major source of soft errors is the cosmic neutron or alpha particle strikes in the circuits [11], [25], [26], the acoustic sensor based detection focuses on the physical characteristic of a particle strike. When a particle strikes the silicon, it produces a large amount of electron-hole pairs that generate phonons and photons in sequence. The resulting phonons and photons spreading out of the striking site make an intense sound wave traveling over the silicon at the speed of l0km/s [27]. In light of this, the particle strike, (the cause of a soft error), can be detected by measuring the change of capacitance of a cantilever beam structure. Such a single acoustic sensor can detect a strike 5*mm* away within 500*ns*, and each sensor roughly takes an area of one square micron [28]. Upasani *et al* [28] show that the entire pipeline can be protected by covering the out-of-order core with a mesh of acoustic sensors, causing only less than 1% area overhead without increasing the number of metal layers during the manufacturing process.

B. Region-Level Soft Error Verification

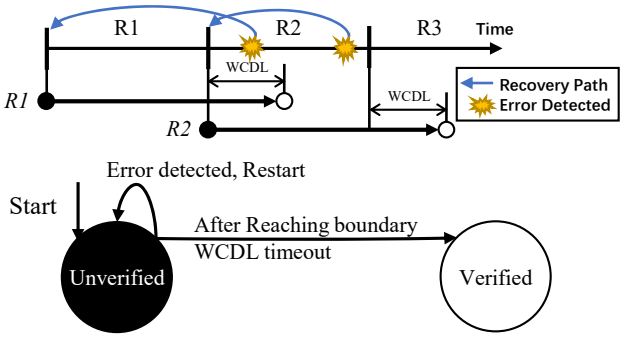

Figure 1. Region verification and recovery timeline (top) and the state change of a region under the verification (bottom)

The beauty of acoustic sensors is that they can verify the absence of soft errors. Given that an error must be detected within WCDL, program execution till any time point will turn out to be error-free WCDL cycles later after that point, unless any error is detected during the WCDL cycles. With that in mind, a couple of prior works, i.e.,

Turnstile [29] and Turnpike [30], realize so-called region-level error verification for out-of-order and in-order CPUs, respectively. During program execution, each region has two states: verified and unverified; see the bottom of Figure 1. Each region starts running in an unverified state. Due to the detection latency of acoustic sensors, the region remains unverified until the WCDL time is passed after the region end (boundary); see the top of Figure 1. Finally, the finished region is to be verified, provided no errors are detected while it waits for the WCDL.

To hide the verification delay of WCDL between regions, the prior works, Turnstile [29] and Turnpike [30], rely on a gated store buffer [31], [32], [33]. They both (1) partition program into a series of regions so that the store buffer never overflows in each region and (2) hold all its committed stores in the buffer until the region is verified. As shown in the top of Figure 1, upon the termination of a region, the following region can be immediately executed without waiting for the verification of the prior region—since the stores of both regions are buffered anyway until they are verified. The store buffer also plays a critical role in recovery; if an error is detected, all the unverified stores in the buffer are discarded, and program control is redirected to the most recently verified region boundary to recover from the error.

Flame adopts the region-level soft error verification on top of acoustic sensors as with Turnstile [29] and Turnpike [30]. However, unlike the CPUs, the target of the prior works, GPUs lack a store buffer, which makes both Turnstile and Turnpike unusable for GPU soft error resilience. The implication is that Flame should look for a different recoverable region formation and find a new way to hide the verification delay (i.e., WCDL).

C. Idempotent Recovery and the Region Formation

To resolve this issue, Flame opts for idempotent processing as an alternative region formation. Idempotent processing is a lightweight region-based recovery scheme that has been used on GPUs for various purposes including register file protection, speculative execution, and exception support [24], [34], [35], [36], [37], [38], [39]. A region of code is idempotent if it can be re-executed multiple times and maintain the same correct output [40]. Thus, it is possible to correct any soft error occurred in an idempotent region by simply restarting the faulty region from the beginning.

The idempotent recovery requires program to be partitioned into a sequence of idempotent regions. To achieve idempotence, a region cannot contain any anti-dependence [41]—also known as Write-After-Read (WAR) dependence—on register/memory variables, which would otherwise cause the input read to be changed in the region making its re-execution non-idempotent [40]. The only exception is when the anti-dependence is preceded by another write to the same variable, forming a Write-After-Read-After-Write (WARAW) dependence that can be included in a region without breaking idempotence. That is because what is overwritten by WARAW dependences is not the input of the region, i.e., they never lead to the input change that is the fundamental reason for non-idempotence.

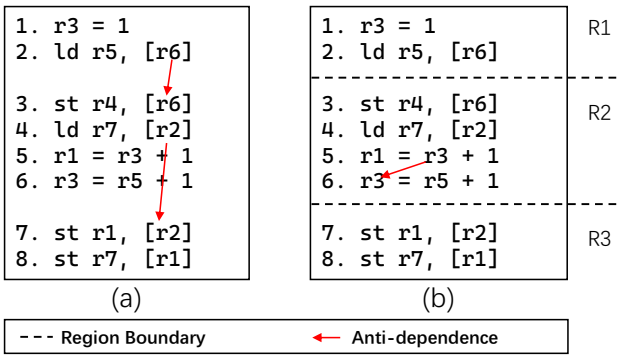

Figure 2. (a) Memory anti-dependence; (b) Initial region partitioning example and the resulting register anti-dependence

1) Memory Anti-dependence: For example, in Figure 2 (a), two instructions pairs 2-3 and 4-7 have anti-dependences on the memory locations addressed by r6 and r2 respectively, and thus the code is not idempotent. To eliminate such memory anti-dependences, region boundaries should be placed to break them, e.g., one before instruction 3 and another before 7. Figure 2 (b) shows the initial region partitioning result where no region has a memory anti-dependence therein.

2) Register Anti-dependence: Besides memory anti-dependences, register anti-dependences also break idempotence and thus need to be eliminated as well. For example, in Figure 2 (b), an instruction pair 5-6 has a register anti-dependence on register r3. Note that this anti-dependence is brought by the region boundary between the first two regions, i.e., R1 and R2; if the instructions 1, 5, and 6 were all in the same region forming a WARAW dependence, the r3's anti-dependence (i.e., WAR dependence) between 5 and 6 would be clobbered by the preceding write of the instruction 1 which prevents the anti-dependence from breaking idempotence.

In general, there are two different techniques for addressing register anti-dependence, i.e., anti-dependent register renaming and live-out register checkpointing. The former eliminates register anti-dependence, and the latter circumvents it—while they both require memory structures to be protected via ECC.

Anti-dependent register renaming renames every anti-dependent register [39] to get rid of the dependence. At instruction 6 of Figure 3 (a), an anti-dependent register r3 is renamed to r8—with renaming the following uses (if exist) as well—to prevent r3's original value from being overwritten. Thus, the register renaming ensures that none of register inputs (e.g., r3 in the figure) is updated, making the resulting region idempotent; all the inputs of the region

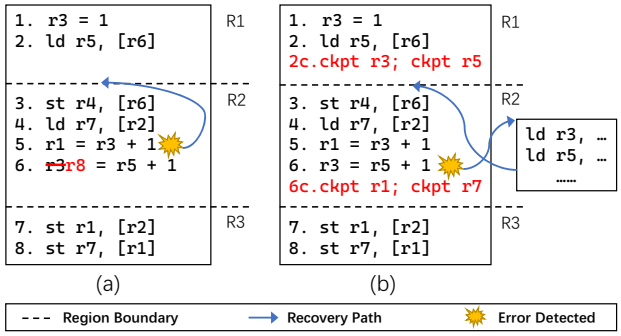

Figure 3. (a) Anti-dependent register renaming; (b) Live-out register checkpointing

R2 can maintain their original values, when it restarts upon error detection, and generate the same designated outputs.

One issue with register renaming is the potential for increasing register pressure, which can lead to more register spills. Nevertheless, our empirical investigation shows that register renaming does not cause a significant execution time overhead (Section VI).

Live-out register checkpointing is another technique for addressing register anti-dependence to achieve idempotent recovery. The state-of-the-art technique [34] inserts a checkpoint (essentially store instruction) to save the value of any updated register in memory, provided it is live-out [41]. For example, in Figure 3 (b), region R1 has two checkpoint instructions at 2c for live-out registers r3 and r5 while region R2 has another two at 6c to save the values of live-out registers r1 and r7. When an error is detected in region R2, its inputs (e.g., r3 and r5) are first restored using the checkpointed values from memory; then, the recovery runtime redirects program control to the beginning of R2 as shown in Figure 3 (b). In this way, R2 can restart with its original inputs and correctly recover from the error[4].

In theory, the live-out register checkpointing can be used as an alternative to the anti-dependent register renaming—if it causes too many register spills thereby degrading the performance. While the register checkpointing does not cause additional register spills unlike the renaming, the checkpoint store often causes significant performance degradation due to the lack of a store buffer in GPUs and requires dedicated memory space for the checkpoint storage. Our choice between the register renaming and the register checkpointing is deferred to Section III-A.

[4]Even if the value being checkpointed is already corrupted by errors (e.g., in pipeline's ALU logic blocks), it is to be used for the re-execution of some later region(s), not the current faulty region where the error occurred. Here, the inputs of the faulty region, which are necessary for its re-execution, are sure to have been checkpointed by some prior region(s) verified to be error-free because of region-level soft error verification (Section II-B).

D. Challenge of Sensing-Based Error Detection Latency

Conceptually, combining acoustic sensor based detection and idempotent recovery can achieve region-level soft error verification in that the sensors can detect all the errors and idempotent recovery can correct all of them. Although the concept is easy, there is a challenge that must be addressed for Flame to realize the region-level verification. For correct recovery, each idempotent region must ensure the absence of errors before moving on to the next region. Otherwise, errors may be propagated to the next region, corrupt its inputs, and eventually render its re-execution non-idempotent. That is why prior idempotent processing techniques require so-called *in-region error detection* [24], [34], [35], [36], [37], [38], [39].

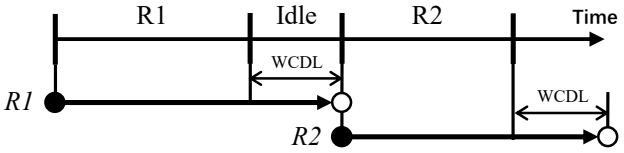

Figure 4. Timeline of region-level verification with idempotent regions and the detection latency of acoustic sensors in consideration; shaded and white circles state *unverified* and *verified* respectively as in Figure 1

However, for an error occurred in each region, acoustic sensors could fail to achieve the in-region error detection due to their sensing delay. Thus, it is critical to verify each finished region by waiting for the WCDL at the region end, but this effectively delays the execution of the next region thereby causing a significant performance overhead as shown in Figure 4. To overcome this region verification delay, Flame exploits GPU's characteristics and proposes WCDL-aware warp scheduling, which can effectively hide the delay and reduce the performance overhead down to almost zero.

III. Flame Approach

The goal of Flame is to achieve lightweight soft error resilience for GPUs so that their entire pipeline can be protected with minimal hardware change. To this end, Flame leverages acoustic sensors and idempotent recovery for soft error detection and recovery, respectively. The Flame compiler partitions the entire GPU program into a series of idempotent regions (with register renaming), while the Flame architecture (1) runs the regions with the region-level verification enforced between them and (2) orchestrates the warp executions to recover from an error upon the detection. In particular, to address the performance issue of waiting for the WCDL verification delay at each region end (i.e., boundary), Flame proposes WCDL-aware warp scheduling for overlapping the delay with other warp's execution (Section III-C) and GPU-specific optimization for extending the idempotent region size and thus requiring fewer verifications (Section III-E). Figure 5 shows this workflow of Flame at a high level.

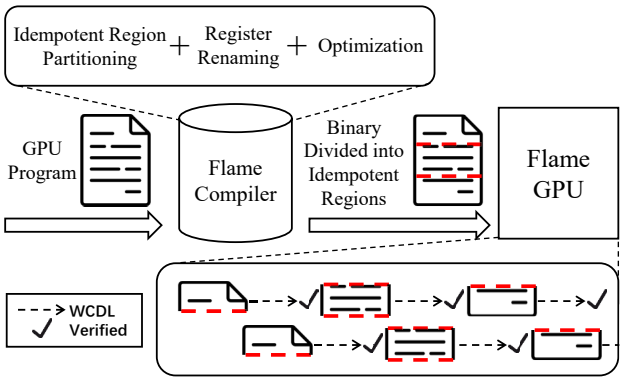

Figure 5. The high level view of Flame

A. Idempotent Region Formation with Register Renaming

Flame leverages idempotent recovery to realize region-level soft error verification and needs to choose an efficient approach to resolving register anti-dependences in the regions where all memory anti-dependences are cut by region boundaries. As discussed in Section II-C2, both register renaming and register checkpointing approaches have their own advantages and limitations. Thus, we conducted an empirical evaluation to evaluate the performance of both approaches and found out that the register renaming would be the more efficient scheme—in spite of the potential performance issue with additional register spills.

Fortunately, GPU's huge register file is very much capable to handle a few additional registers introduced to break the anti-dependences, thereby relieving the register spilling problem. It turns out that register renaming incurs almost negligible performance impact for 34 GPU benchmark applications tested. That is, all of them yield virtually the same occupancy as the baseline that has no soft error resilience support (Section VI). Given all this, Flame chooses register renaming to resolve register anti-dependences.

B. Fault Model and Sensor Deployment

Flame targets radiation-induced soft errors—caused by the strikes of cosmic rays and alpha particles—in order to protect the GPUs from the resulting bit flip errors. Since acoustic sensors detect all particle strikes, Flame can afford to correct both single-bit and multiple-bit upsets without increasing the detection cost determined by the number of sensors being deployed and their network topology. Given that caches and register files on modern GPUs (from Fermi onwards) are already protected by ECC [16], Flame only needs to protect the vulnerable pipeline logic to realize full soft error resilience for GPUs. Consequently, Flame applies the meshes of acoustic sensors to cover the pipeline logic that takes less than half of the total GPU die area. In particular, with 200 sensors per SM, Flame can achieve 20 cycles WCDL with less than 0.1% area overhead on Nvidia's GTX480. Section VI shows different configurations of sensor deployment and their corresponding WCDL.

C. WCDL-Aware Warp Scheduling

For correct idempotent recovery, it is essential to verify that each region is error-free before starting the next region. As shown in Figure 4, the region-level verification demands the worst-case detection latency (WCDL) delay at each region end (i.e., boundary). Since program is split into many short regions, the verification delay increases the execution time to a large extent. While prior work for CPUs [29], [30] can immediately start regions by holding their stores in a store buffer until they are verified (Section II-B), it is not applicable to GPUs due to their absence of a store buffer. Thus, Flame would suffer a significant performance overhead by waiting for WCDL cycles at every region boundary, unless the verification delay is tackled efficiently.

To solve the problem, Flame makes a couple of key observations: (1) The verification delay makes hitting the region boundary no different from a long latency instruction such as load. (2) GPUs can hide such a long latency with their massive warp-level parallelism, e.g., if a warp stalls due to a load miss, the scheduler takes the warp out from the streaming multiprocessor (SM) and schedules one of the other warps there—that are ready to proceed. In light of this, Flame proposes WCDL-aware warp scheduling that lets the scheduler treat idempotent region boundaries as a regular long latency instruction for them to trigger the warp switching—naturally hiding the verification delay piggybacking on GPU's original scheduling mechanism. Note that the scheduling turn-around time of a warp is usually much longer than the verification delay (e.g., 20 cycles of WCDL). Therefore, with the help of the WCDL-aware warp scheduling, Flame can perfectly hide the verification delay—provided there are enough warps to schedule.

D. Hardware Support for Idempotent Recovery and WCDL-Aware Warp Scheduling

Traditional GPU architecture does not have any support for idempotent recovery or WCDL-aware warp scheduling. To facilitate idempotent recovery, Flame devises a recovery PC table (RPT) to hold the recovery PC for each warp. For the WCDL-aware warp scheduling, Flame introduces a region boundary queue (RBQ) in the existing warp scheduler so that it can track all warps' verification and scheduling status. Figure 6 sketches the architecture diagram of the Flame GPU with its newly added microarchitecural components shown in black boxes.

1) Recovery PC Table for Correct Recovery: Acoustic sensors cannot tell exactly which one of the warps running on the SM encounters a soft error or which instruction is corrupted. To ensure that errors are correctly recovered, Flame should perform error recovery for every warp in the

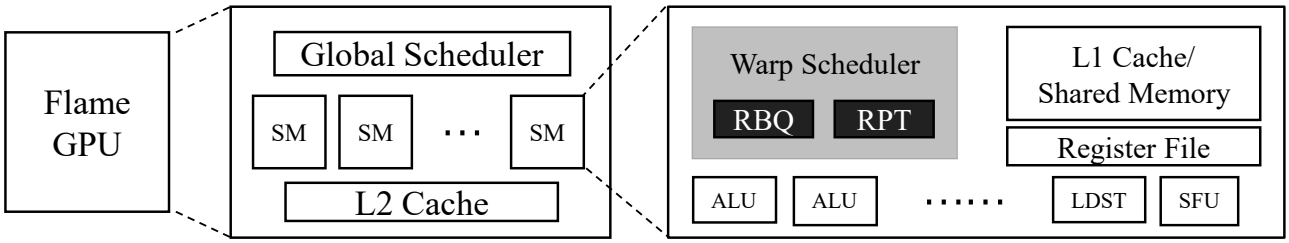

Figure 6. Overview of Flame GPU architecture; RBQ and RPT are the newly added components.

SM upon their error detection. With the help of the region-level verification, warps can recover from a soft error by resuming the execution following their most recently verified region boundary. For this purpose, Flame needs to record the region boundary information for every warp to ensure correct recovery.

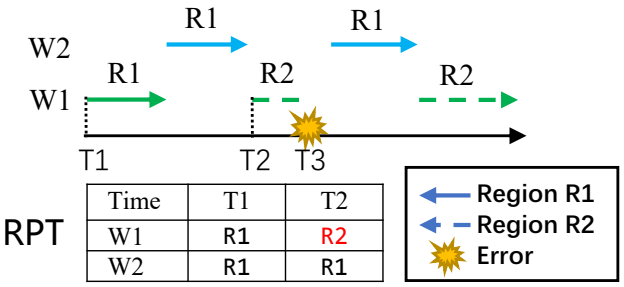

Figure 7. Example of recovery (re-execution) with RPT

To achieve this, Flame adds a recovery PC table (RPT) to the original warp scheduler. For each warp, RPT stores the recovery PC—that is initialized to the beginning of the corresponding warp in the first place. Then, whenever a new region is verified, the recovery PC is updated with the instruction next to the verified region boundary, i.e., the beginning of the youngest region that is yet to be verified. If an error is detected, Flame sets the PC of all warps to their recovery PC stored in the corresponding RPT entry, which effectively starts the re-execution recovery for all warps.

Figure 7 shows an example of the recovery protocol. Here, two warps (W1 and W2) are assumed to run their first two regions R1 and R2 in the SM. At time T2, W1's first region R1 is verified (while the other warp W2 is in the middle of its first region), and RPT sets W1's recovery PC to the beginning of the second region R2. When a soft error is detected at time T3, W1's PC is overwritten with R2—since R1 has already been verified since T2—whereas W2's PC is updated with R1. Consequently, W1 starts R2—following the end of the most recently verified region R1—but W2 restarts the interrupted region R1. Note that RPT should be able to accommodate the number of maximum possible warps (typically 32) because each entry of the table should maintain the recovery PC for each of the warps.

2) Region Verification Conveyor: To achieve WCDL-aware warp scheduling, Flame needs to switch out warps that hit the region boundary and make those warps, that are done with waiting for verification delay, ready to be scheduled back again. Therefore, Flame must (1) track the amount of time each warp has waited for its region to be verified and (2) check the time is greater than the cycles of WCDL, the verification delay. A simple way to track the waiting time would be employing a cycle counter. However, there are many warps in an SM and every warp needs one counter. If Flame used such a naive approach, it would require too many counters causing a significant logic area overhead. With that in mind, Flame proposes the idea of verification conveyor that can precisely track all the warps' verification status with minimal overhead. To shed light on the idea, the following uses a simple analogy.

Figure 8. An illustrative example of verification conveyor

The verification conveyor is like the conveyor oven used in pizza stores and a warp is like a pizza dough. In this analogy, a warp that is waiting for verification delay is analogous to a pizza dough that is being baked. A single conveyor oven can bake as many pizzas as possible, and similarly, one verification conveyor can track as many warps in the SM. When a warp hits the region boundary, it is put into the verification conveyor. Like the conveyor oven, the verification conveyor will move continuously at the pace of one unit per cycle. By setting the conveyor's length to the cycles of WCDL, the warp once placed at the conveyor will come out of its belt as verified—just like a piece of fully baked pizza that is ready to be packaged. Thus, with only one verification conveyor, Flame can ensure that every warp waits for exactly WCDL cycles; and of course the verification order is guaranteed. Figure 8 shows an illustrative example of the verification conveyor.

To implement verification conveyor, Flame adds a region boundary queue (RBQ) structure to the original warp sched-

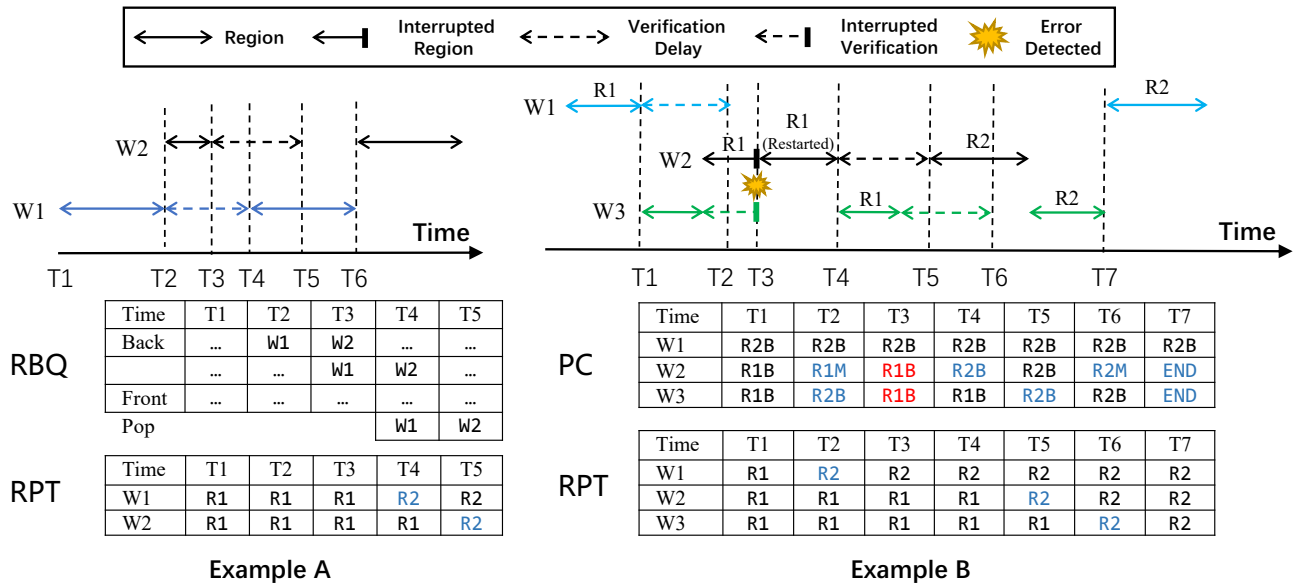

Figure 9. An illustrative example of Flame's hardware support and recovery upon error

uler. RBQ has the length of WCDL and the width of 6 bits (5 bits for 32 warps and 1 bit for validity; see Section VI for more details). Each RBQ entry contains a warp ID and a valid bit. When the warp scheduler encounters a region boundary for each warp, it enqueues a new entry for the warp with the ID into RBQ and sets the valid bit of the entry to 1. The warp is then taken out of the active warps pool and thus becomes ineligible for execution until it is verified (i.e., popped out of the RBQ). At every cycle, the warp scheduler dequeues an entry from RBQ. If the entry is valid, the warp scheduler puts the warp back to the active warps pool, and the RPT entry corresponding to the warp is updated—since it is just verified—with the beginning of its next region. When an error is detected, all entries in the RBQ are discarded. That is because every warp is to be re-executed for recovery by referring to its RPT entry and jumping back to the recovery PC therein.

3) Examples: To show how Flame updates RBQ and RPT over time, Figure 9 provides two illustrative examples: (1) an error-free case and (2) an error case with recovery. For simplicity, we assume only one warp can run at each cycle. The entries in RPT represent the beginning address of region x (Rx) for the corresponding warp, and each warp has two regions R1 and R2. Solid lines represent region execution for a warp while dotted lines represent verification delay (WCDL).

Example A shows the error-free execution with 2 warps W1 and W1. Here, RBQ only shows meaningful entries. In the beginning, RPT has all recovery PC of the warps set to the beginning of their first region R1, and RBQ has no valid entry.

- At T1, W1 starts executing.
- At T2, W1 hits the region boundary, so the scheduler puts W1 into RBQ for verification and schedules W2 to execute.
- At T3, W2 also hits the region boundary, and W2 is put into RBQ for verification. At the moment, W1 is somewhere middle of RBQ. Between T3 and T4, the GPU is idling, since all 2 warps are in the RBQ.
- At T4, W1's entry pops from the RBQ, completing the verification of the warp. The scheduler then updates W1's recovery PC in RPT to the beginning of W1's R2. Since W1 is now verified, the scheduler schedules W1 again to execute its second region R2.
- At T5, W2 becomes verified, and the scheduler updates its RPT entry.
- At T6, W1 ends. The scheduler immediately schedules W2 since W2 has already been verified.

Example B shows the execution with an error detected and the recovery process for 3 warps W1, W2, and W3. In the PC table in Figure 9, RxB means the beginning of region x, and similarly RxM means the middle of region x. In the table, any PC updates due to region execution progress are highlighted in blue while the updates made for recovery are highlighted in red.

- At T1, W1 hits the region boundary, and its entry is put into RBQ for verification.
- At T2, W1 finishes the verification of its first region R1 with its entry popped from RBQ (not shown in the figure), and the scheduler updates its recovery PC in the corresponding RPT entry with R2's beginning—since R1 is just verified. At the moment, W2 is executing

R1, and W3 is waiting for the verification of its already finished region R1.

- At T3, an error is detected while W2 is running in the middle of R1, and W3 is still waiting for the verification. The scheduler starts the re-execution of all warps by resetting their PCs. Since W1 finished verification for its R1, its PC must be reset to R2's beginning, which is already the case since the prior region R1 has been verified since T2. At the moment, due to the error detected, W2's execution is interrupted while W3's verification is invalidated. For both warps, their first region R1 have not been verified yet, and thus their PCs are reset to R1's beginning. After resetting PCs of the warps, the scheduler chooses W2 to execute it; W2 re-executes its interrupted R1.
- At T4, W2 reaches the region boundary of the region R1, and this time W3 is picked by the scheduler to run; W3 re-executes its R1 that was finished once but unable to be verified, and this still ensures correct execution thanks to the beauty of idempotence. After that, Flame follows the error-free execution workflow.

E. Optimization for Extending Idempotent Region Size

In the original idempotent region formation algorithm, all synchronization primitives, e.g., barriers and atomic instructions, are treated as region boundaries in addition to each function call boundary [29], [30], [34], [35]. Along with the region-level soft error verification, this prevents errors occurred in a warp from escaping into other warps across such a synchronization primitive; this is so-called synchronization-level error containment.

However, we observe that such synchronization region boundaries sometimes prevent the compiler from identifying WARAW dependence and result in many small idempotent regions. The implication is that the smaller the regions are the more frequent region boundaries encounter waiting for the region verification. This could put significant pressure on the WCDL-aware warp scheduling, which can lead to noticeable performance overhead, e.g., more than 10% for *LUD* (see Section VI-B2). Across all benchmark applications tested, we find out that synchronization region boundaries for barriers[5] could be safely removed to enlarge the region size, thereby improving the performance due to the less frequent region verification. Since this optimization relaxes the synchronization-level error containment, we conservatively apply it only to the certain code pattern exemplified below for correct idempotent recovery.

1) Background: Barriers in GPU program work within one thread block. Each barrier blocks any warps in a block until all the warps in the block reach the barrier [42], [43]. Therefore, barriers are often used to order the load and store operations from different warps.

[5]We keep boundaries for both other synchronization primitives and those barriers that do not comply with the optimization code patterns.

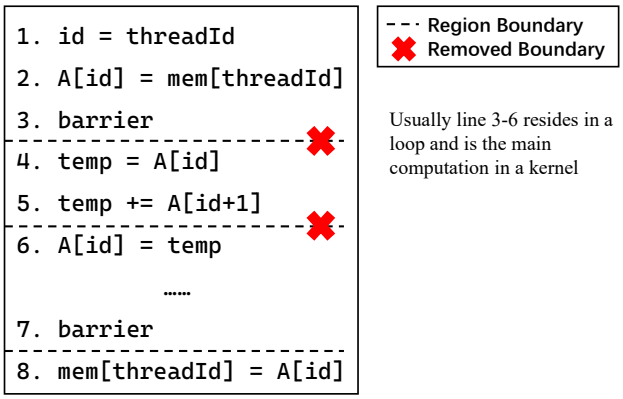

Figure 10. Barrier usage example

Figure 10 shows a general and simplified example of the barrier usage in a GPU kernel. Every warp first initializes shared memory (i.e., **A[id]**) with some value. After the barrier, every warp performs calculations that depend on other warps' data and store the data back to the shared memory. Here, the barrier is used to prevent the warps from accessing uninitialized data and outputting incorrect value.

2) Optimization Method: In the code pattern shown in Figure 10, the original region formation algorithm considers the barrier as a region boundary [29], [30], [34], [35]. However, it is clear that the references to the memory location **A[id]** form WARAW dependence, and therefore the instruction sequence 1-6 can be considered idempotent. Because of the barrier-induced synchronization region boundary inserted after instruction 3, the original idempotent processing compiler is unable to identify such WARAW dependence and ends up inserting another region boundary before instruction 6 due to the anti-dependence (i.e., WAR) on **A[id]** between instructions 4 and 6. By ignoring this barrier-induced region boundary, both region boundaries can be removed.

As a result, soft errors could be propagated across the barrier due to lack of its corresponding region boundary and the region verification. To correctly recover from such errors, Flame takes a conservative optimization policy for bounding the error propagation to the warps in the same block. That is, Flame's compiler only identifies a code section with the following particular pattern: 1) A piece of shared memory is initialized, and all the following memory anti-dependences in the code section result from the piece of shared memory; 2) There must be no memory write to other memory locations within the entire code section. If Flame's compiler finds such a target section, it removes all the barrier-induced synchronization region boundaries in the section and restarts the anti-dependence analysis therein. With this optimization, Flame can let the target code section form a single extended idempotent region and improve the

performance thanks to the less frequent region verification.

3) Correct Recovery: By the aforementioned conservative policy, the target code section chosen for the optimization never writes any other locations except the shared memory initialized at the beginning of the section. Since shared memory data can only be shared by the warps of the same block [43], soft errors cannot escape the block boundary thanks to the above code pattern. In other words, the only possible error propagation case is that data corrupted in a warp end up being read by other warps in the same block. Flame can always handle this case correctly with the help of its recovery protocol that restarts all the unverified regions of the warps in the same block— as one block is only assigned to one SM and Flame performs error recovery for all warps in the SM (Section III-D1) [43].

IV. Discussion

Additional Protection Requirement: For Flame to achieve correct soft error recovery, it is required that the RBQ, the RPT, and the warp scheduler logic blocks should be protected against soft errors. As with prior work [22], [29], [30], [37], [44] and Intel's commodity RAS (reliability/availability/serviceability) processors, Flame assumes that address generation units (AGUs) are hardened against soft errors to avoid misaddressing loads and stores [45]. Finally, Flame also requires register file (RF) controller logic to be hardened so that registers are correctly addressed all the time. We believe that similar technology of the AGU protection can be applied to the RF controller protection.

False Positive Rate: With proper calibration, acoustic sensors can avoid the detection of those particle strikes that do not cause bit flips, thus being able to reduce the chance of reporting such weak strikes to zero [19]. Despite the calibration, the sensors could yield false positives, since not all particle strikes lead to user-visible output errors because of the so-called bit-masking effect. If the sensors never respond to weak particle strikes, the false positive rate should be the same as the bit-masking rate; Li and Pattabiraman show that typical GPU applications have a 63.5% masking rate [46]—that is a lot smaller than that of CPU applications (≈90%) [4], [47], [48]. According to Tiwari et el's field study for analyzing the resilience of GPUs used in a supercomputer [49], the frequency of GPU (post-masking) failure is 0.5 errors per day. Taking into account the frequency along with the bit-masking rate, we expect that a GPU generates $\frac{0.5}{1-0.685} \approx 1.37$ errors per day. As such, the acoustic sensors are expected to report $1.37 \times 0.685 \approx 0.93$ false positives per day. The implication is that the execution time overhead of the false error recovery is not significant at all considering the small idempotent region size, i.e., 50.23 instructions on average for 34 GPU benchmark applications we tested.

Error Containment With the help of idempotent regions, Flame can safely commit stores without any delay—unlike prior CPU resilience works where stores are not allowed to be merged into the L1 cache for their verification [29], [30]. The rationale is that even if corrupted data stored in an unverified region could be written into cache hierarchy, they will never be read because (1) an idempotent region has no anti-dependence and (2) their causative errors are guaranteed to be detected—before the next region starts—and corrected with the idempotent recovery.

Furthermore, given that Flame targets data-race-free program as with prior (idempotent) recovery works [24], [34], [35], [36], [38], [39], [50], [51], [52], [53], corrupted data can never be read by other threads and peer-GPU/CPU. It is important to note that in data-race-free program, cross-thread dependencies or cross-GPU/CPU dependencies must be ordered with explicit synchronization primitives. As idempotent processing basically treats all synchronization primitives as region boundaries [24], [34], [35], [39], [50], Flame ensures that all data written before the region boundary of such a primitive (e.g., barrier) can only be read after the synchronization region is verified due to the region-level soft error verification. In this way, Flame prevents data corruptions by soft errors from being propagated to other threads and peer-GPU/CPU—as long as necessary synchronization primitives are correctly used for data-race-freedom. The only exception occurs when Flame leverages its region size extension optimization, i.e., errors can propagate across a barrier synchronization primitive. However, as discussed in Section III-E3, this turns out to be harmless for the correctness of the program, and Flame still guarantees the safe idempotent recovery of the errors.

V. Evaluation Environment

This section presents our experimental settings for the evaluation of Flame and introduces the state-of-the-art resilience schemes tested for comparison with Flame. Our evaluation focuses on Nvidia's GTX480, one of the most simulated GPUs in the literature. GTX480 is based on Fermi architecture and equipped with ECC to protect the memory hierarchy, i.e., the register file, caches, and memory. By default, Flame uses 20 cycles of WCDL and the GTO (Greedy Then Oldest) warp scheduler, i.e. the default model of GPGPU-Sim v4.0 [54]. Section VI-B3 offers sensitivity analysis results for various GPU architectures, WCDL settings, and warp scheduler models.

A. Simulation Methodology

While implementing anti-dependent register renaming and live-out register checkpointing requires the register information, Nvidia's PTX assembly uses virtual registers and there is no open-source back-end compiler toolchain that can modify register-allocated assembly; this is why simulators take PTX as a proxy for program binary, and prior work [34], [55] instead performs register allocation on PTX for simulation. Flame's compiler takes the same approach with hacking the

PTX-level register allocation to implement both the register renaming and the register checkpointing.

Table I
BENCHMARKS USED FOR SIMULATION

Suite	Application	Abbr.	Suite	Application	Abbr.
parboil	Single precision Matrix Multiply	SGEMM	rodinia 3.1	back propgation	BP
	Lattice-Boltzmann Method Fluid Dynamics	LBM		breadth-first search	BFS
GPGPUsim bench	Neural network	NN		gaussian elimination	Gaussian
	Laplace transform	LPS		hotspot	Hotspot
	AES entryption	AES		lava Moleculer Dynamics	LavaMD
Cuda toolkit Sample	binomialOptions	BO		LU Decomposition	LUD
	convolutionSeparable	CS		Needleman-Wunsch	NW
	scalarProd	SP		pathfinder	PF
	BlackScholes	BS		SRAD_v2	SRAD
	SobolQRNG	SQ		streamcluster	SC
	fastWalshTransform	WT		CFD solver	CFD
	transpose	Transpose		kmeans	Kmeans
	Discrete Haar wavelet decomposition	DWT		k-Nearest Neighbors	KNN
	sortingNetworks	SN	parboil	3-D Stencil Operation	Stencil
	histogram	Histogram		Two Point Angular Correlation Function	TPACF
NPB	Integer Sort	IS	SHOC	STREAM triad	Triad
	Conjugate Gradient	CG	ALTIS	Giga UPdates per Second	GUPS

All our simulations were conducted on top of GPGPU-Sim v4.0 [54]. We modified the warp scheduler model in the GPGPU-Sim to correctly reflect Flame's hardware modification (Section III-D) and ran the compiled PTX code on the modified GPGPU-Sim. Our GPU applications were collected from Rodiania v3.1 [56], Parboil [57], benchmarks from GPGPU-Sim [58], NPB [59], ALTIS [60], SHOC [61], and samples from CUDA toolkit. Table I shows all 34 benchmark applications tested. Note that the reason why we were not able to test all the applications from each benchmark suite is because some of them use those APIs that the GPGPU-Sim cannot handle or some new CUDA APIs that are not currently supported by our register allocation.

B. Competing Schemes

To evaluate Flame in comparison with the state-of-the-art works, we categorized the soft error resilience schemes based on their detection and recovery schemes and test various combinations of the schemes. For the recovery scheme, we tested idempotent recovery with 2 register anti-dependence handling techniques, i.e., the register renaming and the register checkpointing; to fairly evaluate their performance difference (Section VI-B), we used Penny [34], the state-of-the-art register checkpointing based recovery scheme, with its proposed compiler optimization enabled, e.g., optimal checkpoint pruning, checkpoint coloring and scheduling, and automatic checkpoint storage assignment [34]. For the detection scheme, in addition to acoustic sensor based detection, we tested 1) instruction duplication based detection and 2) hybrid detection combining the other two.

1) Instruction Duplication Based Detection: We used SwapCodes [17] as a reference implementation of instruction duplication based detection. Unlike prior instruction duplication schemes [1], [44], [62], SwapCodes does not have to explicitly compare the outputs of the original instruction and its replica, thus being able to outperform the prior schemes significantly. To detect soft errors without comparing the outputs, SwapCodes pairs the original instruction output register with the ECC code of the replica instruction output register. In this way, if the output registers mismatch, it can be detected by ECC's error checking logic without using explicit compare instructions for checking the outputs. While SwapCodes proposes additional optimizations such as move propagation to eliminate unnecessary duplication, the register file requires another hardware complexity in addition to the swap ECC logic. Thus, we used the plain SwapCodes without such optimizations for instruction duplication based detection throughout our evaluation.

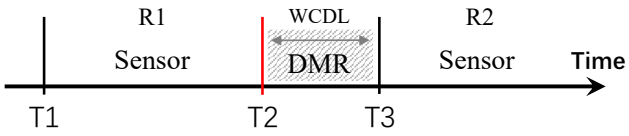

Figure 11. Illustration of tail-DMR

2) Hybrid Detection: Tail-DMR (dual modular redundancy, i.e., instruction duplication) is a hybrid detection technique that opportunistically uses acoustic sensors and instruction duplication with idempotent recovery in mind. [35], [63]. It leverages a different way to hide the verification delay (WCDL) caused by the detection latency of acoustic sensors. Rather than waiting for WCDL at the end of each idempotent region for verification, Tail-DMR includes the delay as a part of the region execution time for the next region to start with no delay. For this purpose, Tail-DMR divides each region into head and tail and duplicates the instructions of tail so that the post-DMR execution time of the tail is as long as WCDL.

Figure 11 shows the illustration of tail-DMR. The execution of region R1 spans from T1 to T3 while its tail executes between T2 and T3 covering WCDL. To detect soft errors in each region, Tail-DMR uses acoustic sensors in the head (i.e., from T1 to T2) whereas it uses instruction duplication in the tail (i.e., from T2 to T3), thus being called tail-DMR. If errors occur in the head of each region, they are guaranteed to be detected by the sensors before the region ends at T3. On the other hand, if errors happen within the tail of the region, which is protected by DMR, then it can immediately detect the errors. Either way, all errors are guaranteed to be detected within the region where they occurred, which eliminates the verification delay between regions.

However, tail-DMR incurs a significant performance overhead due to the DMR (i.e., instruction duplication) overhead at the tail of each idempotent region. To this end, we

implemented the DMR using SwapCodes-based instruction duplication and integrate the hybrid detection (i.e., Sensors + instruction duplication) with both idempotent recovery schemes (i.e., the register renaming and the register checkpointing).

VI. Experimental Results

A. Hardware Cost

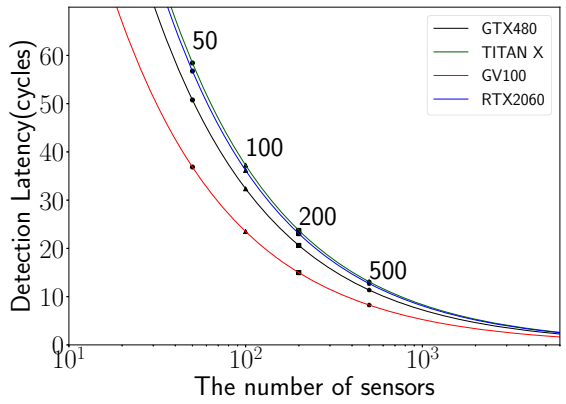

Figure 12. WCDLs when varying the number of sensors per SM

1) Acoustic Sensors: GTX480 runs at 700*Mhz* and has the die size of 512mm^2. By measuring the die shot, all 16 SMs, except the register file and caches, take about 55% of the die area, and each SM takes 17.5mm^2 [16]. Taking the same estimation as prior work [22], we find that 50-300 deployed acoustic sensors can achieve 50-15 cycles of WCDL and, by default, use 200 sensors to achieve 20 cycles of WCDL. Every acoustic sensor costs around 1μm^2 area, and a mesh of 200 sensors with the interconnection network takes much less than 0.01mm^2 [23]. This implies that the total area overhead of deploying the acoustic sensor mesh to every SM is less than 0.1%. Figure 12 shows how the WCDL (Y-axis) varies depending on the number of sensors per SM (X-axis) for 4 different GPU architectures.

2) Warp Scheduler Modification: In GTX480, each SM can hold a maximum of 64 active warps. The SM has two warp schedulers, each of which manages 32 warps [43]. Therefore, one entry of RBQ occupies $5+1=6$ bits, and the size of the entire RBQ becomes $20\times 6=120$ bits. Finally, the size of RBT should be $32\times 32=1024$ bits. Thus, the total cost of Flame's hardware modifications is trivial, and it is important to note that all of them are off the critical path of the GPU.

B. Performance Evaluation

1) Overall Performance Overhead: This section analyzes the execution time overhead for Flame and combinations of different detection/recovery schemes in comparison to the baseline that does not have any resilience support; we tested the following schemes with our default settings, i.e., 20 cycles of WCDL, GTO, and GTX480.

- **Sensor+Renaming** is our full resilience solution (supporting both detection and recovery) using Flame with its idempotent region optimization enabled.
- **Sensor+Checkpointing** is a full resilience scheme that uses the sensor based detection and idempotent recovery with register checkpointing.
- **Renaming** is a recovery-only solution based on idempotent recovery with register renaming.
- **Checkpointing** is another recovery-only solution based on idempotent recovery with register checkpointing.
- **Duplication+Renaming** is a full resilience scheme that detects errors using SwapCodes' instruction duplication and recovers with register renaming based idempotent processing.
- **Duplication+Checkpointing** is a similar full resilience scheme based on SwapCodes error detection and register checkpointing based idempotent recovery.
- **Hybrid+Renaming** is another full resilience scheme comprised of (1) hybrid detection of acoustic sensors and SwapCodes' instruction duplication for tail-DMR and (2) idempotent recovery with register renaming.
- **Hybrid+Checkpointing** is a similar full resilience scheme based on the same hybrid detection and idem-

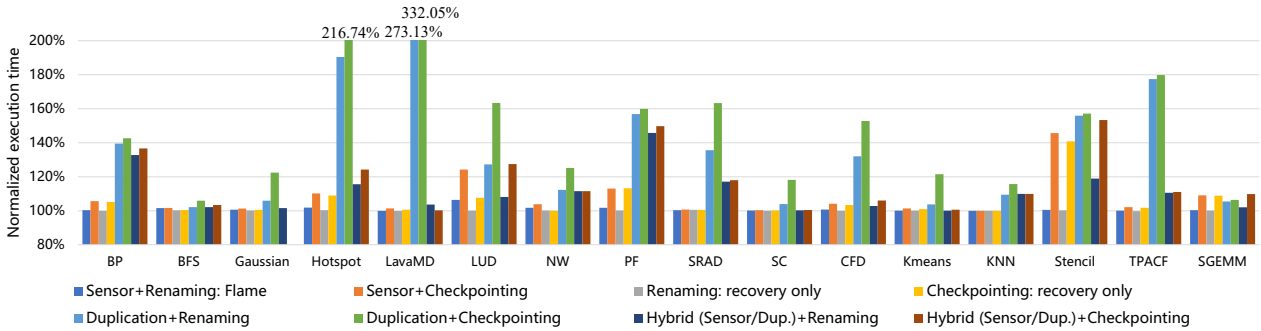

Figure 13. Performance overhead of Flame and other schemes with 20 cycles of WCDL and the GTO scheduler on GTX480

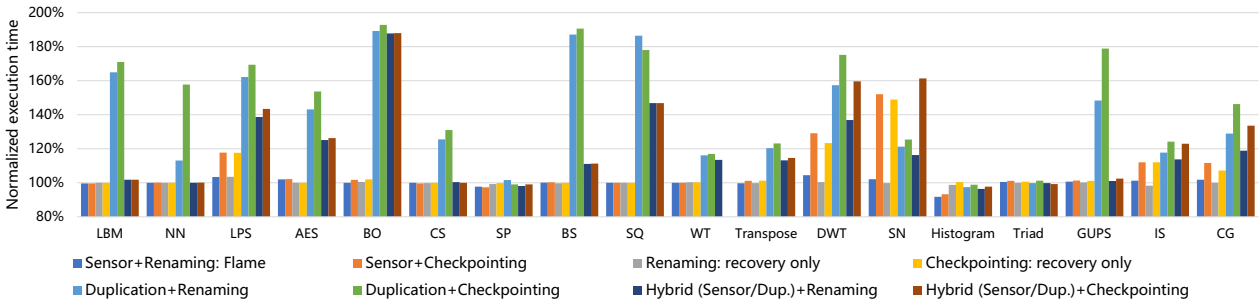

Figure 14. Performance overhead of Flame and other schemes with 20 cycles of WCDL and the GTO scheduler on GTX480 (cont.)

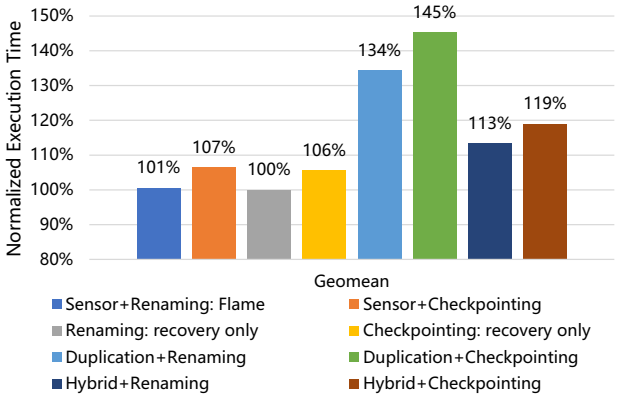

Figure 15. Average performance overhead of Flame and other schemes with 20 cycles of WCDL and the GTO scheduler on GTX480 (geomean)

potent recovery with register checkpointing.

For the above schemes, Figure 13 and 14 both show their execution times normalized to the original applications without soft error resilience. Figure 15 presents the geometric mean of the normalized execution times of all 34 benchmark applications tested. We only showed fault-free execution time overheads because the soft error rate is typically very low, i.e., 1.37 errors per day (Section IV). As a result, the fault-free execution overhead is a more important metric for evaluating soft error resilience schemes. It is worth noting that the average region size for all the benchmarks is 50.23 instructions, and thus the recovery overhead for re-executing 50 instructions within 24 hours is negligible.

As shown in Figure 15 **Renaming** incurs only 0.04% overhead on average and up to 3.5% for *LPS* (Figure 14). The impact of register renaming turns out to be trivial due to rare register anti-dependences found in GPU applications; for most applications, the performance of **Renaming** is almost the same as the baseline. In contrast, **Checkpointing** incurs 5.9% execution time overhead on average and up to 40.8% for *Stencil* (Figure 13); the slowdown is mainly caused by checkpoints (i.e., essentially store instructions) inserted in innermost loops, due to the lack of a store buffer.

Meanwhile, the execution time overheads of **Duplication+Renaming** and **Duplication+Checkpointing** are 34.38% and 45.34%, respectively. Despite the compare-free nature of SwapCodes, its instruction duplication still incurs a significant performance overhead. On the contrary, an acoustic sensor based detection scheme as in Flame and **Sensor+Checkpointing**, incurs little to negligible overhead. The execution time overhead of **Sensor+Checkpointing** is 6.9% on average, and it is only a little higher than **Checkpointing**. Similarly, Tail-DMR also benefits from the efficiency of acoustic sensor based detection. **Hybrid+Renaming** and **Hybrid+Checkpointing** cause 13.48% and 19% average overhead, respectively. While the execution times are much better than those of **Duplication+Renaming** and **Duplication+Checkpointing**, the DMR part still incurs a considerable overhead compared to acoustic sensor only detection.

Finally, Flame outperforms all other schemes significantly and only causes 0.6% execution time overhead on average and up to 6.4% for *LUD* (Figure 13). The slowdown originates from region boundaries inside a loop of a small but frequently executed kernel. In Figure 14, two interesting results are found in (1) *Histogram* where Flame achieves 8.3% performance improvement over the baseline and (2) *SP* where it does 2.3% performance improvement. We suspected that the improvements result from the different warp scheduling behavior caused by Flame's modification (Section III-D). In particular, Histogram shows much fewer memory bank conflicts (15% reduction compared to the baseline) while *SP* generates fewer L1 cache misses.

2) Impact of the Idempotent Region Optimization: Figure 16 shows the performance improvement achieved by Flame's idempotent region optimization III-E. We found that due to the conservative optimization policy, only 7 benchmark applications can benefit from the extension of idempotent regions. Most notably, for *LUD*, the execution time overhead is reduced from 15% to 6.4%. Similarly, the execution time overhead of *CG* is reduced from 9.7% to 1.7%. Overall, with the help of the region size extension optimization, Flame reduces the performance overhead of the 7 benchmark applications from 4.8% to 1.7% on average.

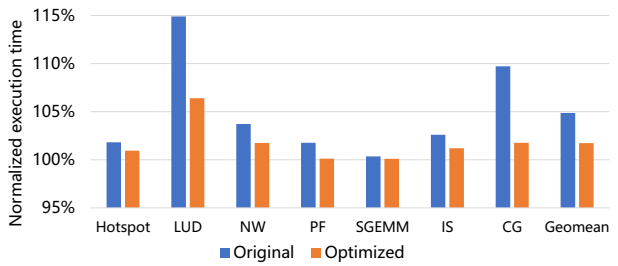

Figure 16. Impact of idempotent region optimization with 20 cycles of WCDL and the GTO scheduler on GTX480

3) Sensitivity Analysis: This section evaluates Flame's performance sensitivity to different WCDL settings, scheduler models, and GPU microarchitectures.

Sensitivity to different cycles of WCDL: We tested different configurations of WCDL ranging from 10 cycles to 50 cycles. Figure 17 describes the results for each WCDL. The overall trend is that the smaller the WCDL is, the lower the overhead of Flame is. When WCDL varies between 10 and 50 cycles, Flame's average execution time overheads ranged from 0.13% to 2.1%. Although the best performance is achieved with 10 cycles of WCDL, this configuration requires 5 times more sensors for each SM than what is required with 20 cycles of WCDL. That is why Flame picks the 20 cycles as its default WCDL, and we believe that it is the more cost-effective choice.

Sensitivity to different scheduler models: We tested three more different warp scheduler models provided by GPGPU-Sim 4.0, i.e., OLD, LRR, and 2-Level, in addition to Flame's default scheduler GTO. Figure 18 shows their performance overhead results with 4 different baselines of each scheduler model with no resilience support. GTO runs a single warp until it stalls, and then picks the oldest ready warp to run [64]. While OLD simply picks the oldest ready warp to run at every cycle, LRR (Loose Round-Robin) schedules warps in a round-robin order at each cycle and skips any stalled warps. Finally, 2-Level divides warps into two groups and applies LRR on one of the groups until all warps in the group stall. Overall, Flame achieves near-zero performance overheads for all 4 scheduler models tested. Although 2-Level shows the highest average overhead, it is only 1.58% on average. Similarly, the average performance overheads of LRR and OLD are also negligible, i.e., 0.76% and 1.18%, respectively. Note that GTO turns out to outperform other schedulers in our evaluation, and again Flame's overhead for GTO is only 0.6% on average.

Sensitivity to different architectures: We also tested three more recent GPU architectures on GPGPU-Sim 4.0. RTX2060 is the latest architecture that is currently supported by the GPGPU-Sim. Figure 19 depicts the normalized execution times of the same benchmark applications when they execute on TITAN X, GV100, RTX2060, and GTX480 in comparison to the 4 baselines of running the original applications on top of each GPU architecture; here, all architecture configurations use 20 cycles of WCDL. Overall, for the 3 new GPU types, we observed the same performance trend shown in the old architecture GTX480. The average performance overheads of the new GPU types all stay less than 1% (geometric mean) while the highest overhead is found for TITAN X, i.e., only 0.97% overhead.

Table II
THE NUMBER OF SENSORS REQUIRED TO ACHIEVE 20 CYCLES OF WCDL FOR DIFFERENT GPU ARCHITECTURES

	Core Frequency	SM Counts	Sensors per SM	Area Overhead
GTX 480	700	16	200	$< 0.1\%$
RTX 2060	1365	30	248	$< 0.1\%$
GV 100	1136	80	128	$< 0.1\%$
TITAN X	1000	24	260	$< 0.1\%$

Newer GPUs typically feature a larger die size and higher core clock frequency, thereby impacting the overhead of acoustic sensors deployed on the GPUs. Table II shows the specification of the four GPU architectures and the number of sensors per SM required to achieve 20 cycles of WCDL.

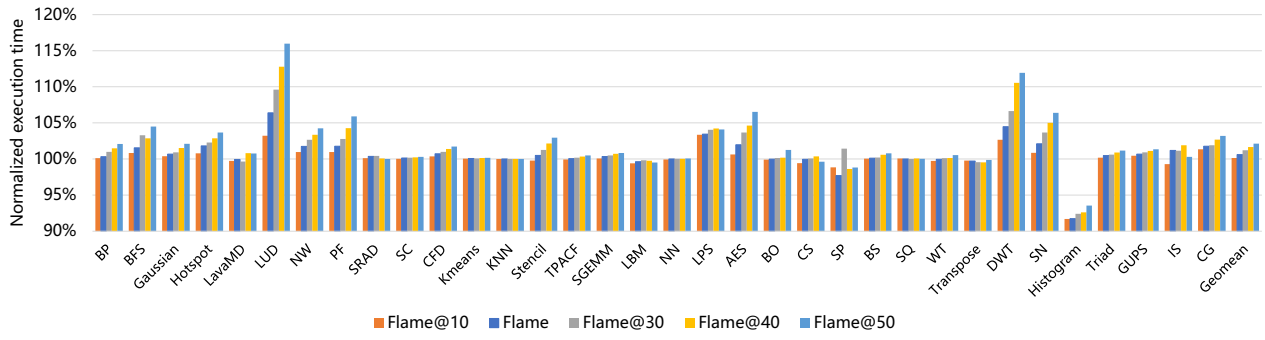

Figure 17. Performance overhead of Flame when WCDL varies from 10 to 50 with the GTO scheduler on GTX480

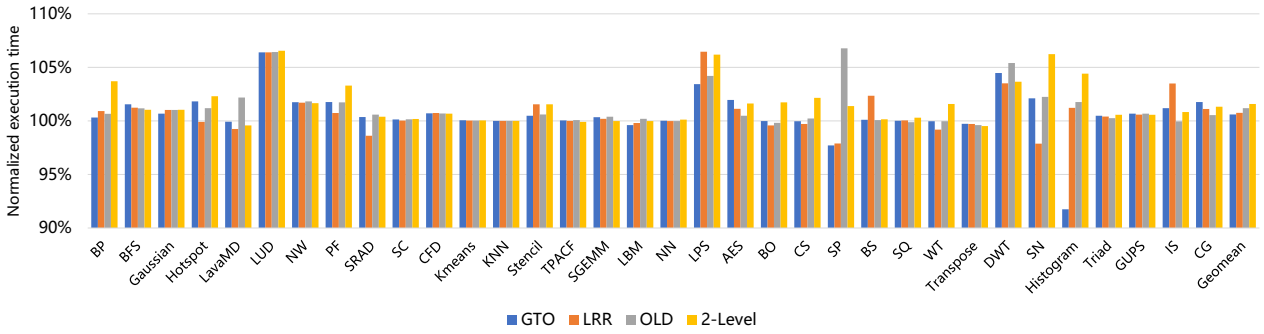

Figure 18. Performance overhead of Flame for different scheduler models with 20 cycles of WCDL on GTX480

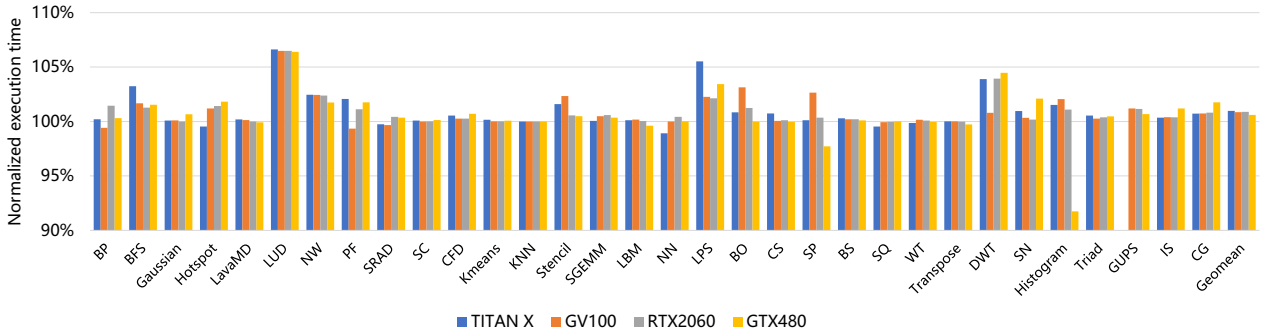

Figure 19. Performance overhead of Flame for different GPU architectures with 20 cycles of WCDL and the GTO scheduler

VII. Related Work

A. Software Approaches for Soft Error Protection

To achieve soft error resilience, many software based approaches take advantage of redundant execution [44], [48]. In general, they duplicate the instruction stream of program and schedule the replicas—on the same core—along with the originals to detect any instruction output mismatch. In this line of research, SWIFT, the seminal error detection work, selectively replicates instructions excluding loads and stores with ECC memory assumed and executes them with unused instruction-level resources of the processor core pipeline [44]. The following variants of SWIFT all take the same instruction duplication approach—though they are different in terms of the sphere of replication (SoR) provided. For example, nZDC [62] increases the error detection coverage by extending the SoR to virtually all kinds of instructions including load, store, branch, and call instructions. Later, the authors of nZDC propose NEMESIS [65] to extend the fault coverage of the existing error correction scheme that relies on majority voting to correct SWIFT-detected errors. While NEMESIS can recover from 97% of detected errors, it comes with a 3X slowdown on average.

The major problem of these approaches is that they incur a significant performance overhead due to the dynamic instruction count increase for satisfying required fault coverage. In contrast, Flame achieves a near-zero overhead. That is because its error detection relies on acoustic sensors that are off the critical path of the instruction pipeline. Also, Flame's idempotence-based recovery is very lightweight on GPUs. Furthermore, Flame can effectively hide the region-level verification delay with its WCDL-aware warp scheduling leveraging GPU's massive warp-level parallelism.

B. Hardware Approaches for Soft Error Protection

Considering the performance overhead of the software approaches, researchers exploit hardware resources for redundant multi-threading for fast and comprehensive soft error resilience. Ainsworth and Jones show how heterogeneous computer architecture—comprised of a high-performance out-of-order core and multiple low-power in-order cores—can be used to offer parallel error detection without significant performance overheads [66]. The same authors later extend their work to provide parallel error correction on top of the heterogeneous computer architecture [67]. In case the error occurred in a core is propagated to other cores damaging them, the authors leverage logging-based rollback recovery for the cores to recover from the error.

Other researchers come up with the idea of selectively protecting only a subset of values, e.g., highly susceptible data [68], [69]. Kim and Somani observe that a cache line in the MFU (most frequently used) or the MRU (most recently

used) position is more susceptible to soft errors than those in other positions. With that in mind, the authors propose to replicate such cache lines through the cache hierarchy [68]. The selective protection idea is further explored by Zhang's replication cache [69]. It also duplicates the cache line in the MRU position but saves it into a small additional cache. Compared to aforementioned hardware approaches, Flame requires minimal hardware modification by piggybacking on the existing warp scheduler of GPUs and does not compromise the error protection capability since the acoustic sensors offer complete soft error detection.

C. Hybrid Approaches for Soft Error Protection

For cost-effective soft error resilience, there are hardware/software co-designed proposals. Shoestring is one of the most popular co-designed schemes which combines compiler analysis and symptom-based fault detection hardware to reduce the number of instructions to be duplicated [48]. The key insight is that there is no need to duplicate those instructions whose fault can be detected anyway by the symptom-based detection hardware. Thus, the Shoestring compiler duplicates only statistically-vulnerable instructions—that tend to manifest as user-visible errors without leading to fault symptoms such as exceptions, page faults, overflows [48]. Once such vulnerable instructions are duplicated, they are validated by comparing the results of the original instruction and the replica as with traditional instruction duplication approaches. Nonetheless, Shoestring duplicates still a large number of such vulnerable instructions, which is the main reason for its significant performance overhead. Moreover, Shoestring is a detection-only scheme thus not being able to correct detected errors. In contrast, Flame replicates no instruction since the error detection is made possible with acoustic sensors. The only instruction overhead of Flame is loads/stores for stack-spilled variables caused by the register renaming—though they are rarely found in our experimentation.

D. Task-Level Idempotency Based Resilience

Recent works adopt task-level idempotency to achieve soft error resilience for GPUs or accelerators. SPITS is a programming model for computing partially idempotent tasks [70]. It provides a set of APIs and a runtime system that allow programmers to generate, commit, and manage idempotent tasks on a large number of computing nodes. Likewise, Asymmetric Resilience is another task-level idempotency approach that attempts to achieve soft error resilience for heterogeneous systems [71]. Basically, task-level idempotency allows each idempotent task to be re-executed—when an error is detected therein—for recovery.

The key advantage of the task-level idempotency is that it is applicable to various architectures, e.g., non-Von Neumann architecture accelerators [71]. However, the problem of the above approaches is that not all applications are written with the task-level idempotency in mind. Whenever encountering non-idempotent tasks, Asymmetric Resilience requires heavyweight checkpoints, that save the entire memory space used by the task for backup/recovery, thereby causing a significant performance overhead [71]. Similarly, SPITS needs other fault-tolerant mechanisms to protect non-idempotent tasks [70]. In contrast, Flame's compiler can enable idempotent recovery for any GPU applications.

Finally, slow recovery is another problem of task-level idempotency approaches whereas Flame's recovery is fast due to the small region size (Section IV). Unfortunately, the recovery overhead of re-executing the entire task is very significant, which makes it unrealistic to combine acoustic sensors and task-level idempotency. Although acoustic sensors can detect all radiation-induced soft errors, they may yield some amount of false positives unless the sensors are appropriately calibrated. Even in such a case, Flame can still perform well because of its negligible recovery overhead whereas the task-level idempotency approaches can suffer a significant overhead due to the large task execution size or even stagnation [72], [73], [74], [75].

VIII. Conclusion

This paper presents Flame, a hardware/software co-design scheme that can achieve featherweight soft error resilience for GPUs. Flame uses acoustic sensor based detection for 100% coverage of radiation-induced soft errors along with idempotent processing for their fine-grained recovery. Across idempotent regions, Flame exploits so-called soft error verification, that waits for worst-case detection latency (WCDL) of the sensors at each region end, to verify the absence of errors during the region execution before starting the next region. In particular, to hide the WCDL verification delay, Flame piggybacks on GPU's inherent warp-level parallelism, i.e., descheduling any warps that hit a region boundary—as if it were a regular long-latency operation—and switching to another ready warp. Since the verification delay is generally shorter than the warp's scheduling turnaround time, Flame can effectively hide the verification delay of each region. To track the verification timing and recovery information of warps, Flame devises a region boundary queue and a recovery PC table that can be realized without significant hardware cost. The experimental results demonstrate that Flame incurs negligible average performance overheads for 4 different warp schedulers, outperforming the state-of-the-art resilience schemes. The upshot is that Flame can fully protect GPUs against radiation-induced soft errors without significantly increasing the execution time or the hardware complexity.

Acknowledgment

We thank the anonymous reviewers and our shepherd for their valuable comments. This work was supported by NSF grants 2001124 (CAREER), 1814430, and 2153749.

REFERENCES

[1] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing Software-Directed Instruction Replication for GPU Error Detection," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 842–854.

[2] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards A Systematic Framework for Instruction-Level Approximate Computing and its Application to Hardware Resilienc," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–14.

[3] F. F. dos Santos, S. K. S. Hari, P. M. Basso, L. Carro, and P. Rech, "Demystifying GPU Reliability: Comparing and Combining Beam Experiments, Fault Simulation, and Profiling," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 289–298.

[4] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost Program-level Detectors for Reducing Silent Data Corruptions," in *IEEE/IFIP international conference on dependable systems and networks (DSN 2012)*. IEEE, 2012, pp. 1–12.

[5] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling Soft-Error Propagation in Programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 27–38.

[6] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Application Resiliency Analyzer for Transient Faults," *IEEE Micro*, vol. 33, no. 3, pp. 58–66, 2013.

[7] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "SASSIFI: An Architecture-level Fault Injection Tool for GPU Application Resilience Evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2017, pp. 249–258.

[8] Q. Liu, "Compiler-Directed Error Resilience for Reliable Computing," Ph.D. dissertation, Virginia Tech, 2018.

[9] S. K. S. Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi, "Gang Error Simulation for Hardware Resiliency Evaluation," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, Jun. 2014, pp. 61–72, iSSN: 1063-6897.

[10] A. R. Anwer, G. Li, K. Pattabiraman, M. Sullivan, T. Tsai, and S. K. S. Hari, "GPU-Trident: Efficient Modeling of Error Propagation in GPU Programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. Atlanta, Georgia: IEEE Press, 2020, pp. 1–15.

[11] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The Soft Error Problem: An Architectural Perspective," in *11th International Symposium on High-Performance Computer Architecture*. IEEE, 2005, pp. 243–247.

[12] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, "Real-World Design and Evaluation of Compiler-Managed GPU Redundant Multithreading," vol. 42, no. 3. ACM New York, NY, USA, 2014, pp. 73–84.

[13] T. Tsai, S. K. S. Hari, M. Sullivan, O. Villa, and S. W. Keckler, "NVBitFI: Dynamic Fault Injection for GPUs," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 284–291.

[14] M. B. Sullivan, N. Saxena, M. O'Connor, D. Lee, P. Racunas, S. Hukerikar, T. Tsai, S. K. S. Hari, and S. W. Keckler, "Characterizing and Mitigating Soft Errors in GPU DRAM," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, pp. 641–653.

[15] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[16] Nvidia, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," Tech. Rep., 2009.

[17] M. B. Sullivan, S. K. S. Hari, B. Zimmer, T. Tsai, and S. W. Keckler, "SwapCodes: Error Codes for Hardware-Software Cooperative GPU Pipeline Error Detection," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 762–774.

[18] K. Naveed and H. Wu, "Aster: Multi-Bit Soft Error Recovery Using Idempotent Processing," *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 4, pp. 928–937, 2017.

[19] G. Upasani, X. Vera, and A. Gonzalez, "A Case for Acoustic Wave Detectors for Soft-Errors," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 5–18, 2015.

[20] G. Upasani, X. Vera, and A. González, "Reducing DUE-FIT of Caches by Exploiting Acoustic Wave Detectors for Error Recovery," in *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. IEEE, 2013, pp. 85–91.

[21] C. Chen, G. Eisenhauer, S. Pande, and Q. Guan, "CARE: Compiler-Assisted Recovery from Soft Failures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–23.

[22] G. R. Upasani, "Soft Error Mitigation Techniques for Future Chip Multiprocessors," Ph.D. dissertation, Universitat Politècnica de Catalunya, 2016.

[23] E. C. Hannah, "Cosmic Ray Detectors for Integrated Circuit Chips," Patent, Dec. 18, 2007, US Patent 7,309,866.

[24] M. De Kruijf and K. Sankaralingam, "Idempotent Code Generation: Implementation, Analysis, and Evaluation," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–12.

[25] R. C. Baumann, "Radiation-Induced Soft Errors in Advanced Semiconductor Technologies," *IEEE Transactions on Device and materials reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[26] F. G. Previlon, C. Kalra, D. R. Kaeli *et al.*, "Characterizing and Exploiting Soft Error Vulnerability Phase Behavior in GPU Applications," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[27] G. Upasani, X. Vera, and A. González, "Setting an Error Detection Infrastructure with Low Cost Acoustic Wave Detectors," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 333–343.

[28] G. Upasani, X. Vera, and A. González, "Avoiding Core's DUE & SDC via Acoustic Wave Detectors and Tailored Error Containment and Recovery," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 37–48, 2014.

[29] Q. Liu, C. Jung, D. Lee, and D. Tiwarit, "Low-Cost Soft Error Resilience with Unified Data Verification and Fine-Grained Recovery for Acoustic Sensor Based Detection," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[30] J. Zeng, H. Kim, J. Lee, and C. Jung, "Turnpike: Lightweight Soft Error Resilience for In-Order Cores," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, pp. 654–666.

[31] A. Klaiber *et al.*, "The Technology Behind Crusoe Processors," *Transmeta Technical Brief*, 2000.

[32] J. Choi, Q. Liu, and C. Jung, "CoSpec: Compiler Directed Speculative Intermittent Computation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 2019.

[33] Q. Liu and C. Jung, "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)*, Aug 2016.

[34] H. Kim, J. Zeng, Q. Liu, M. Abdel-Majeed, J. Lee, and C. Jung, "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*, 2020, pp. 989–1004.

[35] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler Directed Lightweight Soft Error Resilience," in *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, LCTES 2015*. ACM New York, NY, USA, 2015, pp. 2:1–2:10.

[36] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-Directed Lightweight Checkpointing for Fine-Grained Guaranteed Soft Error Recoveryery," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2016, pp. 228–239.

[37] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August, "Encore: Low-Cost, Fine-Grained Transient Fault Recovery," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 398–409.

[38] M. De Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An Architectural Framework for Software Recovery of Hardware Faults," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 497–508, 2010.

[39] J. Menon, M. De Kruijf, and K. Sankaralingam, "iGPU: Exception Support and Speculative Execution on GPUs," vol. 40, no. 3. ACM New York, NY, USA, 2012, pp. 72–83.

[40] M. A. De Kruijf, K. Sankaralingam, and S. Jha, "Static Analysis and Compiler Design for Idempotent Processing," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 475–486.

[41] S. Muchnick *et al.*, *Advanced Compiler Design Implementation*. Morgan kaufmann, 1997.

[42] Nvidia, "Parallel Thread Execution ISA Version 7.6," 2022.

[43] Nvidia, "CUDA C++ Programming Guide," 2022.

[44] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software Implemented Fault Tolerance," in *International symposium on Code generation and optimization*. IEEE, 2005, pp. 243–254.

[45] Intel, "Intel® Xeon® Processor E7 Family:Reliability, Availability, and Serviceability," Tech. Rep.

[46] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding Error Propagation in GPGPU Applications," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 240–251.

[47] N. J. Wang and S. J. Patel, "ReStore: Symptom-Based Soft Error Detection in Microprocessors," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 188–201, 2006.

[48] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic Soft Error Reliability on the Cheap," in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010, p. 385–396.

[49] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux *et al.*, "Understanding GPU Errors on Large-scale HPC Systems and the Implications for System Design and Operation," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 331–342.

[50] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 258–270.

[51] J. Jeong and C. Jung, "PMEM-Spec: Persistent Memory Speculation (Strict Persistency Can Trump Relaxed Persistency)," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 517–529.

[52] J. Jeong, J. Zeng, and C. Jung, "Capri: Compiler and Architecture Support for Whole-System Persistence," in *International Symposium on High-Performance Parallel and Distributed Computing*, 2022.

[53] J. Jeong, J. Hong, S. Maeng, C. Jung, and Y. Kwon, "Unbounded Hardware Transactional Memory for a Hybrid DRAM/NVM Memory System," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 525–538.

[54] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 473–486.

[55] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, and D. Fan, "CRAT: Enabling Coordinated Register Allocation and Thread-Level Parallelism Optimization for GPUs," *IEEE Transactions on Computers*, vol. 67, no. 6, pp. 890–897, 2017.

[56] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[57] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," *Center for Reliable and High-Performance Computing*, vol. 127, p. 27, 2012.

[58] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *2009 IEEE international symposium on performance analysis of systems and software*. IEEE, 2009, pp. 163–174.

[59] S. Seo, G. Jo, and J. Lee, "Performance Characterization of the NAS Parallel Benchmarks in OpenCL," in *2011 IEEE international symposium on workload characterization (IISWC)*. IEEE, 2011, pp. 137–148.

[60] B. Hu and C. J. Rossbach, "Altis: Modernizing GPGPU Benchmarks," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 1–11.

[61] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite," in *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*, 2010, pp. 63–74.

[62] M. Didehban and A. Shrivastava, "nZDC: A Compiler Technique for Near Zero Silent Data Corruption," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[63] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-Directed Soft Error Detection and Recovery to Avoid DUE and SDC via Tail-DMR," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 2, pp. 32:1–32:26, Dec. 2016.

[64] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 72–83.

[65] M. Didehban, A. Shrivastava, and S. R. D. Lokam, "NEMESIS: A Software Approach for Computing in Presence of Soft Errors," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 297–304.

[66] S. Ainsworth and T. M. Jones, "Parallel Error Detection Using Heterogeneous Cores," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 338–349.

[67] S. Ainsworth and T. M. Jones, "ParaMedic: Heterogeneous Parallel Error Correction," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 201–213.

[68] Seongwoo Kim and A. K. Somani, "Area Efficient Architectures for Information Integrity in Cache Memories," in *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*, 1999, pp. 246–255.

[69] W. Zhang, "Enhancing Data Cache Reliability by the Addition of a Small Fully-Associative Replication Cache," in *Proceedings of the 18th Annual International Conference on Supercomputing*, 2004, p. 12–19.

[70] E. Borin, I. L. Rodrigues, A. T. Novo, J. D. Sacramento, M. Breternitz, and M. Tygel, "Efficient and Fault Tolerant Computation of Partially Idempotent Tasks," in *14th International Congress of the Brazilian Geophysical Society & EXPOGEF, Rio de Janeiro, Brazil, 3-6 August 2015*. Brazilian Geophysical Society, 2015, pp. 367–372.

[71] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, Q. Chen, M. Guo, and V. J. Reddi, "Asymmetric Resilience: Exploiting Task-Level Idempotency for Transient Error Recovery in Accelerator-Based Systems," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 44–57.

[72] J. Choi, H. Joe, Y. Kim, and C. Jung, "Achieving Stagnation-Free Intermittent Computation with Boundary-Free Adaptive Execution," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2019, pp. 331–344, iSSN: 2642-7346.

[73] J. Choi, L. Kittinger, Q. Liu, and C. Jung, "Compiler-Directed High-Performance Intermittent Computation with Power Failure Immunity," in *Real-Time & Embedded Technology and Applications Symposium*, 2022.

[74] J. Zeng, J. Choi, X. Fu, A. P. Shreepathi, D. Lee, C. Min, and C. Jung, "ReplayCache: Enabling Volatile Cachesfor Energy Harvesting Systems," in *International Symposium on Microarchitecture (MICRO)*, 2021.

[75] J. Choi, H. Joe, and C. Jung, "CapOS: Capacitor Error Resilience for Energy Harvesting Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2022.