Compiler-Directed Soft Error Resilience for Lightweight GPU Register File Protection

Hongjune Kim
Seoul National University, Korea
hongjune@aces.snu.ac.kr

Jianping Zeng
Purdue University, United States
zeng207@purdue.edu

Qingrui Liu
Annapurna Labs, United States
qingrui@amazon.com

Mohammad Abdel-Majeed
University of Jordan, Jordan
m.abdel-majeed@ju.edu.jo

Jaejin Lee
Seoul National University, Korea
jaejin@snu.ac.kr

Changhee Jung
Purdue University, United States
chjung@purdue.edu

Abstract

This paper presents Penny, a compiler-directed resilience scheme for protecting GPU register files (RF) against soft errors. Penny replaces the conventional error correction code (ECC) based RF protection by using less expensive error detection code (EDC) along with idempotence based recovery. Compared to the ECC protection, Penny can achieve either the same level of RF resilience yet with significantly lower hardware costs or stronger resilience using the same ECC due to its ability to detect multi-bit errors when it is used solely for detection. In particular, to address the lack of store buffers in GPUs, which causes both checkpoint storage overwriting and the high cost of checkpointing stores, Penny provides several compiler optimizations such as storage coloring and checkpoint pruning. Across 25 benchmark applications, Penny causes only ≈3% run-time overhead on average.

CCS Concepts: • **Hardware → Error detection and error correction**; • **Software and its engineering → Error handling and recovery**; *Compilers.*

Keywords: GPU, Resilience, ECC

ACM Reference Format:
Hongjune Kim, Jianping Zeng, Qingrui Liu, Mohammad Abdel-Majeed, Jaejin Lee, and Changhee Jung. 2020. Compiler-Directed Soft Error Resilience for Lightweight GPU Register File Protection. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3385412.3386033

1 Introduction

Due to technology scaling and near-threshold computing [9, 18, 23, 55, 59], soft error resilience has become as important as power and performance in any computing systems. For example, when high-energy particles strike the circuit, they might cause application crashes and even worse, silent data corruptions (SDC) which corrupt the program output without being detected. Near-threshold voltage and process variation make it harder to predict the response of the circuits to a particle strike, thus making them more susceptible to soft errors [5, 18, 23–25, 28, 48, 51, 55, 59].

With the popularity of GPUs, it is becoming more important to protect them against soft errors [22, 56]. The GPUs of all major supercomputers and data centers have already adopted hardware support for soft error resilience. NVIDIA GPUs from Fermi onwards use error correction code (ECC) to protect their storage structures even including register files (RFs). However, ECC-protected RFs do not only increase the critical path of instruction execution but also often lead to a longer clock cycle than ECC-free RFs [4, 39, 40, 57]. Due to the increased delay and power [42], ECC-protected RF consumes significantly more energy than ECC-free RF.

Another big concern for an ECC-protected RF is its area, e.g., 22% overhead for a 32-bit register. The ECC overhead becomes worse for multi-bit errors that commodity GPUs already report. Since they cannot be handled by conventional single-bit error correction and double-bit error detection (SECDED) ECC [63], much more bits should be paid to protect against such multi-bit errors. Along with the combinational logic for encoding/decoding, ECC-protected RFs thus occupy a significant amount of area that could otherwise be used to enlarge RFs/caches thereby improving the performance of GPUs. Given all this, there is a compelling need for lightweight GPU RF protection.

With that in mind, we propose Penny, a new GPU RF resilience scheme that combines recent advances in idempotent recovery [13, 14, 21, 32, 34–36, 41] with error detection code (EDC), e.g., single or multi-bit parity checking. Compared to error correction code (ECC) which imposes high bit-wise data redundancy, EDC [49] used by Penny introduces less area overhead—because EDC only needs to detect errors. The reduced bit-redundancy reduces not only the area overhead but also the access latency and static/dynamic power consumption of RFs. Therefore, Penny achieves the same level of resilience as ECC at a much lower cost.

Alternatively, by paying the same area overhead as ECC, Penny guarantees to detect and correct wider multi-bit errors, thus providing stronger resilience; when a 32-bit register uses 7-bit ECC for 1-bit correction, Penny offers 3-bit correction using the same 7-bits. Since they are used solely for detection as EDC, Penny can detect 3-bit errors. Once errors are detected, Penny's idempotent recovery can correct them no matter how many bits are corrupted.

A region (i.e., instruction sequence) of code is idempotent if it can be re-executed many times and still result in the same correct output [14]. Thus, the program can recover from soft errors by simply restarting the idempotent region where they occurred. Among the existing schemes, Bolt [35] is particularly suitable for our needs, because it does not require the RF to be protected by ECC for correct recovery, unlike other idempotent schemes [12, 13, 15, 21, 32, 34, 36, 41]. To achieve correct soft error recovery without ECC, Bolt checkpoints the live-out registers of idempotent regions.

However, naively applying Bolt to GPU faces several important challenges that must be overcome to achieve ECC-free GPU RF protection. First, soft error detection must be fast enough for correct recovery. The existing idempotent recovery schemes require the enforcement of in-region detection, i.e., errors must be detected within the same region where they occurred. However, such a short detection latency puts high pressure on the underlying detection mechanism.

In addition to reducing the hardware cost of ECC, Penny's EDC-based parity-checking has a unique virtue of not requiring in-region error detection. We prove that even if errors on registers are not detected within the region they occurred, they can be safely recovered in any later region where they are detected by with the help of parity-checking; a faulty register is never propagated to other registers/memory because the error is always detected at the register access time. This obviates the need to use expensive detectors whose latency is short enough to detect errors before a region ends.

Second, since Bolt was made for CPUs, there is no consideration of GPU architectures. For example, the existence of shared/global memories in GPUs demands the right checkpoint storage to be chosen between them. Care must be taken to allocate the resources to threads because the concurrency (i.e., the occupancy of a streaming multiprocessor—SM) can be limited by the resource contention between the threads.

Third, GPU lacks store buffers. Unfortunately, they are required for idempotent recovery to correct soft errors [13, 15, 21, 34–36, 41]. The problem is that checkpointing the live-out registers of a current idempotent region may overwrite the checkpoints stored at some earlier region—which are live-in registers of the current region and thus required for its re-execution—thereby failing to recover from errors. This is not an issue for CPUs because their store buffers can either hold checkpointing stores of each region until its end where they are released to memory or discard them on error detected.

Fourth, due to the lack of store buffers, GPUs cannot effectively hide the store latency for a checkpoint, i.e., essentially a store instruction. That is, the overhead of the checkpointing stores can be high, lengthening the critical path of the GPU's pipeline execution—which is not a problem for out-of-order CPUs where stores are off the critical path most of the time. For example, *binomialOptions*, a benchmark in the CUDA toolkit [47], shows a 26.7% slowdown when only 2 checkpointing stores are added into the inner-most loop.

To overcome the above challenges, Penny proposes a new GPU RF protection that can achieve correct yet performant soft error resilience. As with Bolt, Penny uses compiler-generated idempotent regions for recovery. However, unlike Bolt, Penny does not require the in-region error detection that makes it impossible to use idempotent recovery for lightweight RF protection. Also, we solve both the correctness and performance problems of Bolt due to the lack of store buffers in GPUs. To ensure correct idempotent recovery, Penny leverages register renaming and checkpoint storage coloring. They make it possible to correctly restore all the checkpointed inputs to a faulty region upon recovery. To solve the performance overhead of the checkpointing stores, Penny carefully exploits GPU's shared/global memories for the checkpoint storage in a way to maintain the GPU performance. Furthermore, Penny leverages novel optimization techniques such as optimal checkpoint pruning for unnecessary checkpoint removal without compromising the recoverability. Following are our contributions:

- Penny is the first compiler-directed soft error resilience scheme that protects GPU RFs without expensive ECC protection. Penny provides equal or stronger reliability guarantee with significantly reduced hardware cost.
- For the first time, we show that parity-checking (EDC) can be integrated with idempotent recovery to achieve low-cost GPU RF resilience. With parity-checking, Penny safely recovers from errors without the restriction of the in-region error detection. Any features of GPUs do not bound this, and thus Penny is applicable to other architectures.
- Penny proposes a set of compiler optimizations that ensure the checkpoint correctness and reduce the performance overhead in the absence of store buffers in GPUs.
- Penny incurs only ≈3% run-time overhead on average across 25 benchmark applications.

Finally, we provide Penny's limitation and our future work in Section 9.1.

2 Background and Motivation

ECC uses more extra bits for error correction than EDC, thereby imposing high area/latency/energy overheads. In contrast, Penny leverages idempotent recovery to correct

Table 1. Storage cost required by conventional ECC and Penny for protecting a 32-bit register from 1-3 bits of errors; (n, k) means n-bits are required for encoding k-bits of data.

Error	Conventional ECC		Penny	
1 bit	SECDED (39,32)	21.9%	Parity (33,32)	3.1%
2 bit	DECTED (55,32)	71.9%	Hamming (38,32)	18.8%
3 bit	TECQED (60,32)	87.5%	SECDED (39,32)	21.9%

detected errors, thus obviating the need for the redundant information (bits) encoded in ECC for correction. Instead, Penny uses single or multi-bit parity-checking[1] to detect an error in RFs before it is propagated to other registers/memory. The error detection coding (EDC) required for this is much cheaper than ECC. That is because the number of error-bits ECC can correct is smaller than what it can detect. That is, with the same bit-redundancy budget, the number of error-bits ECC can detect is smaller than what EDC can do.

Table 1 compares the required bit budgets of conventional ECC protection and Penny for protecting a 32-bit register from one to three bits of errors. For single-bit error correction, SECDED (39,32) coding [43] is required for ECC protection—i.e., 21.9% bits overhead due to additional 7 bits—whereas only 1 bit is needed for Penny incurring 3.1% overhead. Although SECDED ECC can detect 2-bit errors, it cannot correct them. Such detected unrecoverable errors (DUEs) force program to be restarted from the beginning.

For a more error-prone environment that uses smaller manufacturing technology—e.g., AMD uses a 7nm process for recent Vega GPUs—or near-threshold computing[2], the demands for multi-bit error correction grow fast in the semiconductor industry. For ECC to correctly recover from 2-bit errors, it must use DECTEC (55,32) [43] coding that requires 23 additional bits for every 32-bit chunk of data. In contrast, Penny can detect 2-bit errors with 6-bit Hamming code [43] and correct them by re-executing the idempotent region where they occurred. For 3-bit error correction, ECC must use TECQED (60,32) [43] coding that requires 28 additional bits, while Penny can use SECDED (39,32) coding paying only 7 bits to achieve the same correction.

Note that when commodity GPUs, equipped with traditional SECDED (39,32) ECC, use Penny, they become capable of correcting 3-bit errors as TECQED (60,32) ECC can but without the high cost; as shown in Table 1, using the same SECDED coding, Penny can correct 3-bit errors whereas ECC can correct only 1-bit errors. Alternatively, for single-bit error correction, Penny can replace the SECDED ECC in commodity GPUs with 1-bit parity, thereby drastically saving the hardware cost without compromising the resilience guarantee. The takeaway is that Penny can provide the same level of RF resilience or stronger resilience under the same coding with the significantly lower area, latency, and energy overheads; Section 7.1 provides detailed measurements.

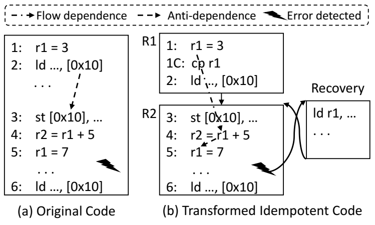
[1]Parity-check is a general term used to signify the process of validating the encoded data, regardless of the used coding scheme [43].

[2]Multi-bit errors increase by 2.6X under near-threshold operations [48].

3 Idempotent Recovery and Challenges

3.1 Idempotent Recovery Overview

An *idempotent region* is a part of the program code that can be freely re-executed and still generate the same correct output. Thus, a program can recover from errors simply by restarting the idempotent region where they occurred. For this reason, researchers have used the side-effect-free re-execution of idempotent regions for many different types of recovery—including misspeculation handling, nonvolatile memory crash consistency, context switching, and power failure recovery [8, 26, 32, 35, 38, 41, 58]. For soft error recovery, Bolt [35] is the state-of-the-art idempotent recovery scheme. Unlike others, Bolt does not require an ECC protected register file for correct recovery. As with Penny, Bolt divides a program into a series of idempotent regions.



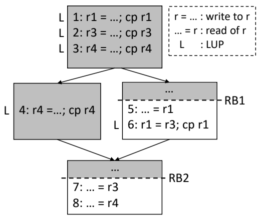
```
1:   r1 = 3
2:   ld ..., [0x10]
     . . .
3:   st [0x10], ...
4:   r2 = r1 + 5
5:   r1 = 7
     . . .
6:   ld ..., [0x10]
```

```
1:   r1 = 3
1C:  cp r1
2:   ld ..., [0x10]

3:   st [0x10], ...
4:   r2 = r1 + 5
5:   r1 = 7
     . . .
6:   ld ..., [0x10]
```

```
ld r1, ...
. . .
```

Figure 1. Idempotent recovery.



```
L 1: r1 = ...; cp r1
L 2: r3 = ...; cp r3
L 3: r4 = ...; cp r4

L 4: r4 =...; cp r4

...
5: ... = r1
L 6: r1 = r3; cp r1

...
7: ... = r3
8: ... = r4
```

Figure 2. Eager checkpointing.

For a region of code to be idempotent, the inputs of the region must not be overwritten, i.e., no anti-dependence [44] on the inputs during the region execution; both memory and register inputs must be preserved to assure the side-effect-free re-execution. Figure 1 shows how idempotent recovery works: (a) is a non-idempotent code that encounters a soft error, and (b) is the transformed idempotent regions. Suppose an input value is passed via memory location 0x10, which is overwritten at line 3 (memory anti-dependence), and the error is detected between lines 5 and 6. One could try to correct it by restarting the code (a) as if it were idempotent, but the value being loaded at line 2 would be different from the original input value. As shown in Figure 1(b), we thus split the code into 2 regions to break every memory anti-dependence, ensuring that memory inputs are never overwritten [15].

Not only that, to guarantee correct re-execution from the beginning of a region $R2$ where the error is detected, but we should also preserve its input registers, e.g., $r1$ is a live-in register of $R2$ in Figure 1(b). Bolt uses *eager checkpointing* to save live-out registers of each region, which are basically live-ins of some following regions. All last update points (LUP) of live-out registers in each region are identified—e.g., line 1 for $r1$ in Figure 1(b)—and their corresponding checkpoint instructions are inserted *right after* LUPs (line 1C). As such, eager checkpointing ensures that for each region being executed, its live-in registers have already been checkpointed. The checkpoint instruction '$cp\ r1$' in the figure is essentially a store instruction that saves the register $r1$ to a dedicated checkpoint storage assigned for each register. When an error is detected in region $R2$, our recovery runtime first restores the register from the checkpoint storage and then redirects the program control to the beginning of the region[3]. That way, correct recovery is assured though $r1$ is overwritten at line 5 in the figure.

Figure 2 shows how the eager checkpointing works in the presence of control divergence. As shown in the shaded part of the figure, an idempotent region can include a conditional branch. Note that a live-in register can have multiple LUPs depending on the control path taken, e.g., $r4$'s values updated at lines 3 and 4 both reach the same region boundary (entry) $RB2$ in Figure 2. Similarly, an updated value at a point can be live-out to multiple region entries, e.g., $r3$ in the figure.

3.2 Idempotent Recovery Challenges for GPUs

Unfortunately, all prior works including Bolt [35] cannot be used for GPUs due to correctness/performance problems.

Checkpoint Overwriting: One issue with Bolt's eager checkpointing is that a checkpoint (i.e., store instruction) in a region can overwrite previously saved checkpoint value while it is still required until the end of the region. In Figure 2, the checkpoint of $r1$ at line 1 is an input to the region beginning with a region boundary $RB1$, but $r1$ is overwritten (at line 6) during the region execution. If an error is detected after line 6 and before the region finishes at $RB2$, the re-execution starting from $RB1$ cannot correct the error. That is because the original value of the region input $r1$—previously checkpointed at line 1—was overwritten and cannot be restored.

To prevent checkpoint overwriting, Bolt relies on hardware called a gated store buffer (GSB) that can hold the checkpointing stores of each region until it finishes; they are eventually merged to checkpoint storage in memory at the region end, provided no error has been detected within the region. Since GPUs lack store buffers, Penny proposes 2 software schemes, i.e., register renaming and storage coloring.

Performance Overhead: The lack of store buffers also has a significant impact on performance overhead of checkpoints that are essentially stores for saving live-out registers. Unlike the CPU where stores are off the critical path in general, they can easily slow down the GPU when the warp-level parallelism is not sufficient to hide the memory latency. This often occurs due to resource limitations on register file and shared memory, suppressing the number of active warps, i.e., occupancy. In reality, merely executing a few more stores can significantly hurt the GPU performance. For example, Bolt's unvarnished adaptation to GPU, for which we only use Penny's automatic assignment of checkpoint storage between shared and global memories, shows 39.0% run-time overhead on average and up to 943.5% (Section 7.3).

Given that soft errors rarely occur (1/day in 16nm [16, 35]), users are reluctant to adopt Bolt for such rare error correction at the cost of paying the high-performance overhead all day. The implication is two-fold from the perspective of Amdahl's law [2]: (1) Penny's optimization should focus on minimizing the fault-free execution time overhead, and (2) the impact of the recovery procedure on the total execution time is negligible due to the low error rate. Unlike Bolt, Penny can effectively shift the run-time overhead of fault-free execution to that of fault-recovery procedure.

4 Recovery without In-Region Detection

All prior idempotent recovery schemes require that errors must be detected within the same region where they occur; due to error propagation behaviors [20, 31], re-executing some later region, where an error is detected, would fail—because the region inputs might have been corrupted by the error. In general, the in-region detection requirement imposes the high cost of implementing the detector that offers such a short detection latency, e.g., expensive software- and hardware-based dual modulo redundancy [50].

However, we found out that when parity-based detection is used for idempotent recovery, the in-region detection requirement is unnecessary. Faulty execution can be safely recovered by re-executing the region where the error is detected, no matter how far the region is from the error occurrence. The reason is two-fold: (1) when parity-checking is used, the corrupted register can never be propagated before it is detected on the first access after corruption. (2) eager checkpointing correctly saves the live-ins required for re-executing the region, even in the presence of errors. The detailed proof can be found in Appendix A. Note that the error detection and recovery do not rely on any distinct feature of GPUs, i.e., our proposed technique can be applied to other types of processors to protect their RF.

5 Overview of Compilation Phases

Penny takes GPU program in the form of PTX code, that is a basis for necessary transformations, and performs several analyses and optimization phases in the following order.

Region formation: Penny partitions the entire program into idempotent regions by breaking every memory anti-dependence to prevent their memory inputs from being overwritten. Penny uses an alias analysis [44] to find all possible memory anti-dependences. For each anti-dependent load/store pair, all execution paths from the load to the store must include at least one region boundary. To minimize the number of region cuts (boundaries), De Kruijf et al. [15] translate the region formation problem into a hitting set problem and use an approximate algorithm; we use it for comparison

[3]More precisely for Penny, when parity mismatch is detected in the region, the exception must be thrown and caught by Penny's recovery runtime; this is another requirement with EDC (parity checking) in GPU's register file. The runtime (1) executes the recovery block that restores live-in registers of the region from checkpoint storage or recovery slice if their checkpoints are pruned (Section 5), and (2) jumps back to the beginning of the region.

with GPU specifics in mind[4]. Once all region boundaries are determined, Penny computes their live-in registers, each of which discovers its last updated points (LUPs).

Preventing checkpoint overwriting: A checkpointed register value can be overwritten before it is used for recovery. To ensure that no necessary checkpoint is overwritten, we introduce two techniques, i.e., register renaming and 2-coloring for storage alternation. Penny also provides an auto-selection mechanism, that can choose the better of the two for a given GPU kernel, by using an instruction-level cost estimation model; Section 6.3 provides the details.

Checkpoint scheduling: While Bolt [35] forces a checkpoint to be placed right after the last update point (LUP) of a register to save it, we found out that the restriction can be relaxed without compromising the recoverability (Section 6.2). With that in mind, we perform a checkpoint scheduling to reduce the estimated cost of inserted checkpoints. We achieve this in 2 steps: one after region formation and the other in code generation. First, we conduct bimodal checkpoint placement; a checkpoint is placed either immediately after the LUP or right before the region end. The later step tunes the bimodal schedule for better performance.

Checkpoint pruning: It is possible to remove a checkpoint provided its value can be recomputed by using other checkpointed values. This phase is to prune such an unnecessary checkpoint whose values can be reconstructed at recovery time by executing a series of other instructions, i.e., so-called *recovery slice*. In a sense, the problem of checkpoint pruning can be formulated as that of finding the *recovery slice* that can recompute the value of the pruned checkpoints. We propose a near-linear-time optimal pruning algorithm that significantly improves both the pruning quality and the solution search time over Bolt's basic pruning algorithm.

Storage assignment and code generation: To save checkpoints, Penny uses two checkpoint storages that are already protected by ECC in GPUs, i.e., shared and global memories. Care must be taken for the storage assignment. Since the low-latency shared memory has a limited size, assigning too many checkpoints there can reduce the GPU occupancy. In light of this, Penny carefully distributes its checkpoints to the two storages, thereby reducing the run-time overhead. Also, Penny leverages an appropriate storage layout with coalesced memory accesses in mind. Finally, during the code generation, Penny performs several compiler optimizations to minimize the added instruction cost due to checkpointing stores and their address calculation. The optimizations include local instruction scheduling, redundant code elimination, and loop invariant code motion, etc.

[4]Penny treats GPU synchronization instructions, e.g., barriers, fences, locks, atomics, as a region boundary, to handle inter-thread anti-dependence. This ensures the correct recovery of data-race-free programs that Penny targets.

6 Performance Optimizations for GPU

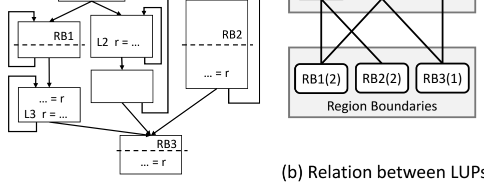
6.1 Checkpoint Cost Estimation

Throughout the optimization process, Penny estimates the cost of each live-out register checkpoint being placed in the program to predict the impact of each optimization. Given that checkpoints in loops lead to significant performance degradation, we focus on removing such checkpoints, especially ones in the inner-most loops. With that in mind, we model the cost of a checkpoint as C^d, where d is the nested-depth of a loop, in which the checkpoint is placed, and C is a constant; we use 64 to prioritize the elimination of a checkpoint in a deeply-nested loop over many checkpoints in a low-depth loop. For a given GPU kernel, we compute its cost by accumulating all costs of checkpoints in the kernel.

6.2 Bimodal Checkpoint Placement

Bolt's eager checkpointing imposes the restriction that all live-out registers of a region must be checkpointed right after their LUPs. However, we found that such a restriction can be safely relaxed, i.e., each checkpoint can be delayed—without compromising the recoverability guarantee—until the region end (boundary). That is because the checkpointed registers in a region are used as inputs to some later regions, not the region itself. This insight allows Penny to schedule checkpoints to minimize the run-time overhead.

However, due to many such possible points in diverse execution paths between LUP and the region boundary, it is indeed a complex problem to achieve the optimal checkpoint scheduling. In light of this, Penny simplifies the scheduling problem with two separate phases. First, for a given live-out register, Penny's bimodal checkpoint placement determines where to place each checkpoint, i.e., either the LUP or the region boundary. The goal of this phase is to identify those checkpoints, that exist inside a loop, and pick them out of the loop. The other phase is performed during code generation to fine-tune the bimodal checkpoint schedule within a basic block level that includes the LUP or the region boundary. This local scheduling considers optimization objectives such as increasing instruction reuse and reducing register usage.



(a) Example program (b) Relation between LUPs and region boundaries

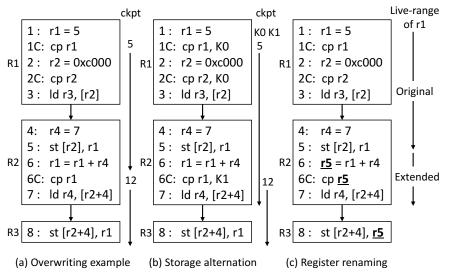
Figure 3. Bimodal checkpoint placement.

In a sense, the bimodal placement is global scheduling in that it picks the checkpoint location between the LUP and the region boundary that can exist across basic blocks. The placement algorithm covers all live-paths—where the checkpoint is used—within the region and minimizes the estimated total cost of the checkpoint to be placed. Figure 3(a) shows how this works with an example control flow graph where a single register r is used for simplicity. Here, r is last updated in 3 different LUPs, $L1$, $L2$ and $L3$.

Note that LUPs and region boundaries have a many-to-many relationship, and thus a checkpoint can be shared between them. For example, if a checkpoint is placed at $L1$, neither $RB1$ nor $RB2$ needs a checkpoint there. Similarly, a checkpoint placed at $RB3$ can obviate both LUPs $L2$ and $L3$. The relation between an LUP and a region boundary can be modeled as a graph where they are represented as vertices. As shown in Figure 3(b), each vertex is labeled by the cost of the corresponding checkpoint. Penny calculates the cost by 2^d, where d is the loop depth. If a register is lastly updated at some LUP, then an edge is introduced from the LUP to the beginning (boundary) of the region to which the register is used as an input.

For each edge in the graph, at least one of the *incident* vertices must be chosen for checkpoint placement, and Penny tries to minimize the total cost of the checkpoints chosen; as shown in Figure 3(b), choosing $L1$, $RB1$ and $RB3$ gives the minimum cost of 4. This problem can be modeled as a weighted version of the *vertex cover problem* that is NP-hard [11] in general cases. However, the problem can be solved in polynomial time in case of a *bipartite graph*—where vertices can be divided into two disjoint sets and all edges connect a vertex from one set to another—as with graphs in our problem. Interestingly, König's theorem [10, 17, 27] shows that the vertex cover problem for a bipartite graph is equivalent to solving the maximum matching of the graph. According to the weighted version of the theorem, Penny uses a maximum-flow algorithm to solve our checkpoint placement in polynomial time.

6.3 Preventing Checkpoint Overwriting



```
1 :  r1 = 5
1C:  cp r1
2 :  r2 = 0xc000
2C:  cp r2
3 :  ld r3, [r2]

4:   r4 = 7
5 :  st [r2], r1
6 :  r1 = r1 + r4
6C:  cp r1
7 :  ld r4, [r2+4]

8 :  st [r2+4], r1
```

```
1 :  r1 = 5
1C:  cp r1, K0
2 :  r2 = 0xc000
2C:  cp r2, K0
3 :  ld r3, [r2]

4 :  r4 = 7
5 :  st [r2], r1
6 :  r1 = r1 + r4
6C:  cp r1, K1
7 :  ld r4, [r2+4]

8 :  st [r2+4], r1
```

```
1 :  r1 = 5
1C:  cp r1
2 :  r2 = 0xc000
2C:  cp r2
3 :  ld r3, [r2]

4:   r4 = 7
5 :  st [r2], r1
6 :  r5 = r1 + r4
6C:  cp r5
7 :  ld r4, [r2+4]

8 :  st [r2+4], r5
```

Figure 4. Checkpoint overwriting prevention.

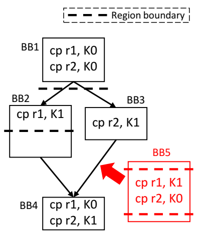


Figure 5. A coloring conflict.

Due to the lack of store buffers in GPUs, a checkpoint storage can be overwritten leading to incorrect recovery. For the example code in Figure 4(a), the value stored in $r1$ at line 1 is a live-in to region $R2$—since it is used at line 5—thus being checkpointed at line 1C. However, the checkpointed value 5 is overwritten by a new checkpoint value 12 at 6C. Thus, if an error occurs between line 6C and the end of $R2$, the original live-in value of $r1$, which is required for restarting $R2$ from its beginning, cannot be restored.

To protect a checkpointed value from being overwritten before it is used for recovery, we introduce 2 software techniques: register renaming and storage alternation. The first technique is to rename the register that causes the checkpoint overwriting. As shown in Figure 4(c), $r1$ is renamed to $r5$, i.e., its checkpoint does not overwrite $r1$'s. To achieve this, Penny artificially extends the live-range of overwritten registers, e.g., Figure 4(c)'s $r1$, in each region until its end [5]. Thus, they exclusively use physical registers, since the register allocator respects the extended live range. This renaming scheme is simple to implement and may save some checkpoint storage spaces compared to storage alternation. However, renaming is likely to increase the register pressure, leading to performance degradation if the register usage becomes the limiting resource of GPU's warp occupancy.

Second, for each overwritten register, Penny maintains a backup storage and alternates the 2 storages. For example, in Figure 4(b), all checkpoints of $r1$ in region $R1$ are saved to storage $K0$ while those in $R2$ are stored to the other storage $K1$, i.e., the value in $K0$ is not overwritten until the end of $R2$. To achieve this, we use a simple 2-coloring algorithm. Applying storage alternation on all registers causes unnecessary storage and run-time overheads. Thus, our compiler first identifies the registers that have at least one checkpoint overwriting and feed them as inputs to 2-coloring for which Penny visits basic blocks in a topological order and colors the checkpoint storages of the input registers. More precisely, if multiple checkpoints of a register exist in each region, which is possible due to a branch in the region, Penny assigns the same color for them. For a given register, Penny flips the color in neighboring regions that checkpoint the same register. Note that such regions are not necessarily consecutive; they can be far away from each other.

The coloring may fail at a control-flow convergence point if the colors from multiple incoming paths are not the same. Figure 5 shows such an example; $r1$'s colors in the 2 paths coming to $BB4$ differ. This causes a coloring conflict. That is, if a left path ($BB2$ to $BB4$) is taken, $r1$'s checkpoint in $BB4$ must be colored with $K0$ since $BB2$ already used $K1$ for $r1$. However, taking the other path ($BB3$ to $BB4$) demands $r1$'s checkpoint in $BB4$ to be colored with $K1$, since $K0$ was used for $r1$ in $BB1$. Thus, the coloring solutions of the 2 paths do not agree with each other. To ensure the same color at the convergence point no matter which path is taken, Penny creates a new adjustment block, that has a dummy checkpoint for a conflicting register, over one or more paths to the point. The goal of a dummy checkpoint is to match

[5]This is similar to what De Kruijf et al. [15] use to deal with register anti-dependences. However, instead of renaming all anti-dependent registers, we rename only live-out registers with anti-dependence. This is particularly beneficial when an anti-dependent register is updated multiple times since Penny only needs to checkpoint the last update, i.e., live-out value.

the same color in other paths. Thus, the color of the dummy checkpoint must be the opposite of the latest color used on top of the new block. As shown in Figure 5, due to a new block $BB5$, the colors in the 2 paths to $BB4$ are both $K1$. Note that the dummy checkpoints are likely to be pruned (Section 6.4), and therefore the resulting overhead is not significant in the majority of applications we tested (Section 7.5).

Also, Penny provides an automatic selection module to choose the better between the register renaming and the 2-coloring based storage alternation. We compile an application using both techniques and estimate their costs in a similar way to the one in Section 6.1 to pick the best.

6.4 Optimal Checkpoint Pruning

Bolt [35] introduced *checkpoint pruning*. The insight is that a large number of checkpoints can be safely *pruned* (removed) without compromising the recoverability guarantee if they can be reconstructed from other checkpointed values available at recovery time. In light of this, Bolt builds the recovery slice (i.e., a series of instructions) of each region to reconstruct its live-in registers whose checkpoints are pruned. Bolt uses a random search to find a possible pruning solution—that tells which checkpoints can be removed. However, the search space dramatically increases as the number of checkpoint increases; the number of possible solutions for n checkpoints is 2^n, i.e., there are 2^n n-bit strings where each bit tells if the corresponding checkpoint can be pruned or not. Thus, instead of validating all possible solutions, Bolt simply finds any first *valid* solution encountered during the random searches, each of which preconceives a random n-bit string solution. The valid solution found is not necessarily optimal in that it is validated as long as the checkpoints corresponding to its set-bit positions can be all pruned. In fact, Bolt ends up leaving many unnecessary checkpoints *committed*, thus causing a significant slowdown in GPUs.

To this end, Penny proposes a novel pruning algorithm that can find an optimal solution with the least estimated cost in polynomial time. Unlike Bolt's search-based approach, Penny validates individual checkpoints by analyzing their dependence from scratch without preconceiving their pruning eligibility, meaning that Penny does not require all pruning decisions to be fixed before validation. Overall, Penny's pruning takes 2 phases. The first phase filters out *trivial* (obvious) checkpoints whose pruning decision turns out to be either *valid* or *invalid* without referring to others. The pruning decision here holds during the entire algorithm, so the next phase simply focuses on the remaining checkpoints whose pruning decision is not finalized by the first phase; we call them *non-trivial* checkpoints. In the second phase, Penny figures out their dependence order, i.e., which checkpoint must be decided before others' pruning decisions due to the dependence. Penny validates the non-trivial checkpoints in the order imposed by the decision dependence to finalize their pruning decisions.

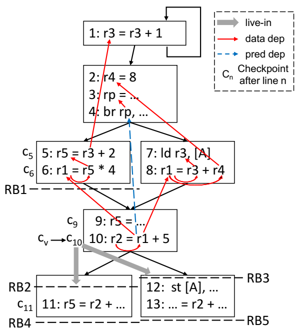


Figure 6. Example of a checkpoint validation.

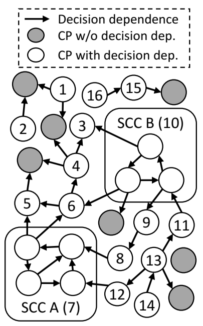


Figure 7. Decision dependence graph.

6.4.1 Phase 1: Filtering Trivial Checkpoints.

To identify *trivial* checkpoints, Penny should validate them first. We use the c_v to refer to a checkpoint being validated and the following rule for its validation.

Rule 1. For c_v to be valid (removable), all the values it depends on must remain the same at the endpoints of all the regions where c_v is used no matter which path is taken to reach the endpoints.

That is because the values must be used for the regions' recovery slice to recompute the value of c_v if it is pruned. In a sense, validating c_v can be understood as building its recovery slice. The validation process requires tracking the necessary dependences over the program's control flow graph. In addition to data dependences [44], Penny considers a new type of dependence called *predicate* dependence. This is necessary when the value on which c_v depends is differently recomputed at control flow paths, e.g., in Figure 6, c_v depends on $r1$ whose value differs across the paths of the branch. Hence, c_v's recovery slice has to include the branch and its predicate, e.g., rp at line 4 in the figure where we say $r1$ is *predicate*-dependent on rp. More precisely, for a value that is defined on multiple paths, it is *predicate*-dependent on the predicates of the branches on which its definitions are control-dependent [44]. We represent predicate and data dependences in a graph and call it PDDG (predicate/data dependence graph).

As shown in Figure 8, Penny validates each checkpoint (c_v) by traversing the PDDG starting from c_v in depth-first search (DFS). The DFS continues by following the dependence chain over the PDDG and terminates at the node whose value can be either safely used at recovery time or dangerous to be used; we call the node a *terminal*. For example, if a register is assigned a constant loaded from GPU's read-only memory, the recovery slice can safely use not only the value by reloading it[6] but also others that only depend on such a *valid* value. Thus, the *validation state* of a PDDG node, i.e., whether its value can be used at recovery time, is determined by those that it depends and their *validation state*.

[6]GPU memory is protected by ECC, and Penny ensures that register file errors never propagate to memory (See Appendix A).

With that in mind, on the way back to c_v where DFS is started, Penny determines the *validation state* of the PDDG nodes visited marking them with one of 3 labels: valid (ϕ_V), invalid (ϕ_I), and undecided (ϕ_U). That is, once *terminal* nodes are marked with either ϕ_V or ϕ_I, the validation state is propagated to their dependent nodes, if necessary, being merged with other states as shown in Figure 8. In particular, when ϕ_I is propagated to a checkpoint node, Penny changes the state to ϕ_U (i.e., undecided). That is because we do not know the pruning decision of the checkpoint yet—if it is committed, the recovery slice could use it. Thus, we simply defer its validation state determination to the next phase and mark it and its dependents with ϕ_U.

Algorithm 1 Marking validation states

```
1:  $\Phi(s)$: Validation state of a PDDG node $s$.
2:  MaxPriority($\phi_a$, $\phi_b$): Higher priority in the order of $\phi_I > \phi_U > \phi_V$.
3:  CheckMemOW($s$, $c_v$): $\phi_I$ if $s$ is overwritten until the endpoints of regions where
    $c_v$ is used, otherwise $\phi_V$.
4:  function MarkValidationStates($c_v$)
5:      return Marking($c_v$, $\{c_v\}$, $c_v$)
6:  function Marking($c_v$, $Visited$, $s$)
7:      if $s \in Visited$ then return $\Phi(s) \leftarrow \phi_I$          ▷ Cyclic dependence found
8:      if $s$ is a constant value then return $\Phi(s) \leftarrow \phi_V$
9:      if $s$ is a load from read/write memory then
10:         return $\Phi(s) \leftarrow$ CheckMemOW($s$, $c_v$)
11:     $\phi_{merged} \leftarrow \phi_V$          ▷ Initialize validation state before merging
12:     $D \leftarrow$ GetPredDataDeps($s$)          ▷ For all predicate/data dependences
13:     for $\forall s_d \in D$ do
14:         $\phi_{dep} \leftarrow$ Marking($c_v$, $Visited \cup \{s\}$, $s_d$)
15:         $\phi_{merged} \leftarrow$ MaxPriority($\phi_{merged}$, $\phi_{dep}$)          ▷ Merge validation states
16:     if $\phi_{merged} = \phi_I$ and $s \in \mathbb{C}$ then          ▷ $\mathbb{C}$: set of all checkpoints
17:         $\phi_{merged} \leftarrow \phi_U$
18:     return $\Phi(s) \leftarrow \phi_{merged}$
19: function GetPredDataDeps($s$)  ▷ $s$ Collect dependences on control flow graph
20:     $D_{data} \leftarrow \{s_d | s \xrightarrow{data} s_d\}$          ▷ $s$ has a data dependence on $s_d$
21:     $D_{pred} \leftarrow \{s_p | s \xrightarrow{pred} s_p\}$          ▷ $s$ has a predicate dependence on $s_d$
22:     return $D_{data} \cup D_{pred}$
```

Algorithm 1 details the validation state propagation process. MarkValidationStates takes a PDDG node c_v as input and calls Mark which performs the depth-first search (DFS) of the PDDG starting from c_v.

DFS terminal condition: The traversal stops at a terminal node and starts to backtrack toward c_v. There are 3 types of terminals: First, the value of the node is constant, i.e., literal or what is loaded from GPU's read-only memory (line 8 in the algorithm). Since it can be retrieved safely, it is marked ϕ_V. Second, any node found in a cyclic dependence chain (line 7), e.g., a loop carried dependence, is terminal, and it is marked ϕ_I due to the difficulty of recomputing the value. Third, a value loaded from memory is also terminal (lines 9-10), and it is valid if it satisfies Rule 1; if the memory value can be used for the recovery of the region where c_v's checkpointed register is used, to reconstruct it, then the PDDG node is marked ϕ_V which is otherwise marked ϕ_I. For example, in Figure 6, c_v checkpoints $r2$ at line 10, and it depends on the memory value loaded from address A at line 7 through the data dependence chain. Here, the memory value must not be overwritten until $RB4$ and $RB5$ because $r2$ is used in the regions ending with these boundaries. However, the intervening store at line 12 overwrites the memory value due to the alias in the address A, and thus the PDDG node of the memory value is marked ϕ_I.

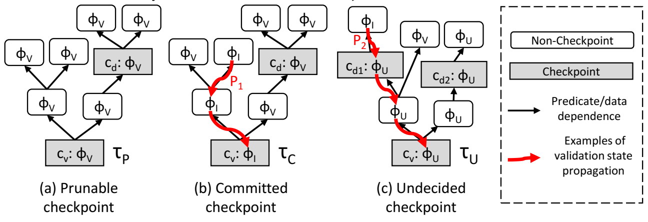


Figure 8. Merging validation states in PDDG.

DFS backtracking and state merging: Once terminal nodes are encountered (lines 7-10 in the algorithm), DFS triggers the backtracking to propagate the validation state of non-terminal nodes being visited to their descendant (lines 11-18). For a non-terminal node, Penny collects all nodes it depends on (line 12) and visits them (lines 13-14). The validation state of the dependent node is determined by merging the state it depends on (line 15), i.e., picking the highest with the precedence of $\phi_I > \phi_U > \phi_V$. The intuition is that for a PDDG node to be *valid*, all the nodes it depends must be valid (Rule 1) as shown in Figure 8(a). In contrast, propagation path P_1 in Figure 8(b) shows that the decision of c_v is dictated by a single terminal node with ϕ_I.

Finally, for a checkpoint node visited, line 16 of the algorithm checks if its input state is ϕ_I; if so, the state is lowered to ϕ_U (line 17). Figure 8(c) shows such an example; on the propagation path P_2, ϕ_I becomes ϕ_U through the intervening checkpoint C_{d1}. A more concrete example is found in the control flow graph of Figure 6. Although $r3$ in line 1 is invalid (ϕ_I) due to the loop-carried dependence, Penny marks the state of its dependent $r5$ (at line 5) with ϕ_U. In this way, Penny leaves a chance for $r5$'s checkpoint, if committed, to be used to reconstruct c_v rather than giving it up by marking the state with ϕ_I.

Once all validation states are merged backed to c_v, Penny uses the resulting state of c_v to decide its pruning decision as one of three: τ_P (pruned) if it is in ϕ_V, τ_C (committed) if it is in ϕ_I, and τ_U (undecided) if it is in ϕ_U. The pruning decisions of τ_P and τ_C are final, and thus only undecided (τ_U) checkpoints move onto the next phase. Our evaluation shows that the first phase can finalize the pruning decisions of the majority of checkpoints, so the second phase only needs to deal with a small number of the remaining undecided checkpoints.

6.4.2 Phase 2: Handling Undecided Checkpoints.

Penny first discovers the dependence between undecided (τ_U) checkpoints. If the pruning decision of one checkpoint is subject to that of another, we say they have *decision dependence* and call its graph representation a *decision dependence graph* (DDG). Then, Penny visits each DDG node (i.e., τ_U checkpoint) in a topological order, finalizing their pruning decision.

Note that the decision dependence naturally imposes an order on the pruning decision between the checkpoints. To guarantee all prerequisite decision results are available before validating a checkpoint, Penny follows the order imposed by the decision dependence to validate and determine the pruning decisions of the remaining checkpoints—starting from the node that only depends on trivial checkpoints whose pruning decisions are already made.

Analyzing Decision Dependence. Suppose the register value stored by c_d can be used for the reconstruction of checkpoint c_v. To realize such a dependence, the 2 conditions have to be satisfied: (1) c_d is committed and (2) all checkpoints that can possibly overwrite c_d until the endpoints of all the regions where c_v's register value is used must be all pruned; see Rule 1. For example, in Figure 6, for c_{10}, that depends on c_5, to be safely pruned, c_5 must be committed, and c_9 and c_{11}, that overwrite c_5, must be pruned. That is, in order to validate c_v, the pruning decisions of c_5, c_9, and c_{11} must be computed beforehand.

Algorithm 2 Computing decision dependences

```
1: $T(c)$: Pruning decision of a checkpoint $c$.
2: OWCkpts($c$, $c_v$): Checkpoints possibly overwriting $c$ until the endpoints of re-
   gions where $c_v$ is used.
3: function CollectDecisionDeps($c_v$)
4:     return GetDecisionDeps($c_v$, $\{c_v\}$, $c_v$)
5: function GetDecisionDeps($c_v$, $Visited$, $s$)
6:     if $s \in Visited$ then return $\{\}$          ▷ Stop if a cyclic dependence is found
7:     if $s \in \mathbb{C}$ and $T(s) = \tau_C$ then          ▷ For a committed ($\tau_C$ ) checkpoint
8:         return OWCkpts($s$, $Exp_{end}(c_v)$)
9:     $F \leftarrow \{\}$                    ▷ Set of nodes $c_v$ has decision dependences on
10:    if $s \in \mathbb{C}$ and $T(s) = \tau_U$ then          ▷ For an undecided ($\tau_U$) checkpoint
11:        $F \leftarrow F \cup \{s\} \cup$ OWCkpts($s$, $Exp_{end}(c_v)$)
12:    $D \leftarrow$ GetPredDataDeps($s$)                          ▷ From Algorithm 1
13:    for $\forall s_d \in D$ do
14:        $F \leftarrow F \cup$ GetDecisionDeps($c_v$, $Visited \cup \{s\}$, $s_d$)
15:    return $F$
```

Algorithm 2 details the dependence analysis. CollectDecisionDeps collects all decision dependences of c_v by traversing the PDDG by following the dependence chain until a committed (τ_C) checkpoint is encountered (lines 7-8). For each committed (τ_C) checkpoint c_d, Penny adds to F (c_v's dependence set) all the checkpoints possibly overwriting c_d until the endpoints of the regions where c_v is used (OWCkpts in the algorithm), according to Rule 1. Note that c_d does not have to be included in the decision dependence because its pruning decision (τ_C) is already made. Pruned checkpoints (τ_P) do not have checkpointed values to use, so they are ignored and Penny advances to the next PDDG dependence. For undecided (τ_U) checkpoints c_d, Penny conservatively considers decision dependence for either case of the checkpoint being pruned/committed. Penny adds the undecided checkpoint c_d and the checkpoints overwriting it (OWCkpts) to c_v's dependence set F at line 11 and continues the depth-first search to encounter a committed checkpoint.

Ordering and Finalizing Pruning Decision. Penny now navigates the decision dependence graph (DDG) obtained from Algorithm 2 in a topological order. Figure 7 shows an example DDG; the colored nodes represent trivial checkpoints, whose pruning decision is already decided in the first phase, and therefore they do not have decision dependence on others.

Except for the nodes with a cyclic dependence, Penny can determine the pruning decisions of all the other nodes by following the reverse order of the decision dependence. Penny uses Tarjan's algorithm [54] to sort the DDG in a topological order along with identifying strongly connected components (SCCs) in a traversal. As shown in Figure 7, Penny then visits and validates DDG nodes in the resulting topological order (i.e., shown as increasing numbers in the figure) to determine their pruning decision; again, such a decision-order-preserving traversal ensures that when each checkpoint c_v is visited, all the necessary validation states of other checkpoints on which c_v depends have already been available.

To validate each checkpoint, Algorithm 1 can be used to traverse the checkpoint's PDDG and obtain a final decision. However, Penny skips the redundant validations by only checking the validity of the nodes in the dependence set of the checkpoint (i.e., F of Algorithm 2).

For an SCC that has a cyclic dependence, which makes the dependence-order-preserving traversal improper, Penny treats all the nodes within each SCC as a single DDG node. This implies that Penny needs to make a pruning decision for all the nodes within an SCC before moving to the next DDG node in the topological order. To find the best combination of the pruning decisions for the nodes within an SCC, Penny performs a brute-force search using the cost model (Section 6.1); we found no SCC in our evaluation. In the absence of SCCs, our overall pruning algorithm has $O(mn)$ time complexity where m is the code size and n is the number of checkpoints in the code.

6.5 Automatic Checkpoint Storage Assignment

To achieve better performance, Penny automatically assigns committed checkpoints to storages by considering both memory access latency and thread-level parallelism in a balanced manner. For checkpoint storages, Penny uses shared memory (in SRAM) and global memory (in DRAM but cached) that are both protected by ECC in GPUs [46]. Shared memory is shared between threads in a thread block and has a limited size. Although shared memory has a significantly lower latency compared to global memory, allocating shared memory over a certain limit can hurt the performance due to diminished warp-level parallelism, i.e., low *occupancy* [46]. With that in mind, Penny first figures out how much shared memory can be used without reducing the occupancy.

Then, Penny scores the live-out registers—whose checkpoints are committed—with the sum of all their checkpoint costs over the entire program (Section 6.1). By taking into account the resulting cost, Penny can prioritize a frequently accessed register over others to allocate it into the low-latency shared memory. That is, Penny tries to assign as many registers as possible to the shared memory before it reaches the occupancy-preserving limit, and then the rest of the registers are assigned to the global memory.

It is important to note that Penny's 2-coloring based storage alternation does not significantly increase memory footprint. The reason is two-fold. First, Penny's 2-coloring only assigns an additional storage into those checkpoints that are overwritten; only a small number of registers (25% on average) require the storage alternation, and it is further reduced by checkpoint pruning. Second, Penny allocates storages only for those registers whose checkpoints are committed at least once. As a result, the average storage size required for each register is only 0.75. That is because Penny's optimal pruning removes the vast majority of checkpoints.

6.6 Low-Level Optimizations and Code Generation

After the checkpoint pruning, Penny performs several low-level optimizations to further reduce the run-time overhead of committed checkpoints. In GPUs, calculating the effective address of the checkpoint storage requires multiple instructions. To reduce the instruction count, Penny conducts a variant of common subexpression elimination, loop invariant code motion (LICM), and induction variable optimization.

Finally, Penny performs local checkpoint scheduling to improve the decision made by the bimodal checkpoint scheduling (Section 6.2). The local scheduling works in a basic block level by pushing down the LUP checkpoints toward the region boundary and pushing up the region boundary checkpoints toward LUP. That is, LUP checkpoints can be placed between their LUP and the end of their corresponding basic block, while region boundary checkpoints can be inserted at any point from their region boundary up to the beginning of the basic block that includes the boundary. In particular, Penny evaluates each possible point to find the best that can maximize the reuse of previously calculated checkpoint address and minimize the register usage.

7 Evaluation

This section evaluates Penny with 2 different ways: (1) hardware logic synthesis and (2) GPU architecture simulation.

7.1 Hardware Cost Evaluation with Logic Synthesis

Table 2. Hardware overheads comparison across RF coding schemes required by ECC protection and Penny (per bank).

	Conventional ECC					Penny				
Err. bits	Coding	Area	Acc. lat.	Acc. energy	Leak. pow.	Coding	Area	Acc. lat.	Acc. energy	Leak. pow.
1b	SECDED	21.9%	25.6%	21.1%	20.7%	Parity	3.1%	3.5%	3.0%	3.0%
2b	DECTED	40.6%	49.2%	39.2%	38.4%	Hamming	18.8%	21.8%	18.1%	17.7%
3b	TECQED	87.5%	74.3%	84.5%	82.7%	SECDED	21.9%	25.6%	21.1%	20.7%

To make a strong case for Penny's production, we designed several register file (RF) coding schemes required for both conventional ECC-based protection and Penny using 22nm with CACTI 6.5 [60]. Our designs assume that the RF is 256KB, and it is divided into 16 banks. We also used Synopsys design compiler [53] to synthesize the built designs for their evaluation. Table 2 shows the overhead of each coding scheme compared to the baseline RF that has no protection; according to synthesis results, the baseline RF has an area of $0.105mm^2$, access latency of $1.01ns$, energy-per-access of $9.64pJ$, and leakage power of $4.7nW$ for each bank. For a single-bit error recovery, SECDED ECC protection incurs 21.9% area overhead while Penny's single parity-bit solution incurs only 3.1% overhead. We observe that the overheads of access latency/energy and leakage power show similar trends, and Penny's low-cost hardware benefits become larger when multi-bit errors should be corrected.

Table 3. Applications used for evaluation.

Suite	Application	Abbr.	Suite	Application	Abbr.
GPGPU-Sim bench [3]	Coulombic potential	CP	Parboil	2-point angular correlation	TPACF
	Libor Monte Carlo	LIB			
	Laplace transform	LPS		SP Matrix multiplication	SGEMM
	Neural network	NN			
	N Queen	NQU	Rodinia [6]	Back propagation	BP
CUDA toolkit samples [47]	Binomial options	BO		Breadth-first search	BFS
	Black-Scholes	BS		Gaussian Elimination	GAU
	Convolution separable	CS		Hotspot	HS
	Scalar product	SP		Molecular Dynamics	MD
	Sobol filter	SQ		Needleman-Wunsch	NW
	Fast Walsh transform	FW		Pathfinder	PF
	Matrix transpose	MT		Speckle reducing anisotropic diffusion	SRAD
Parboil [52]	Sparse matrix-vector mult.	SPMV			
	Jacobi stencil	STC		stream cluster	SC

7.2 GPU Architecture Simulation Setup

The idempotent recovery should be aware of physical register names to ensure the live-in values of regions are safely preserved. Unfortunately, there is no publicly available CUDA toolchain for modifying the register-allocated assembly code and executing it on real GPUs. Thus, simulators such as GPGPU-Sim [3] use PTX code as a basis for the cycle-level simulation, and tools such as CRAT [62] conduct register allocation on PTX code and run it on GPGPU-Sim to study the performance impact of allocated registers. As with CRAT, we allocate physical registers on the PTX code and then apply Penny's transformations on the code. The resulting PTX code is then executed on top of GPGPU-Sim that complies with our register allocation. As the target simulation model, we use Tesla C2050 GPU based on Fermi architecture; the GPU is equipped with ECCs in the RF/cache/memory. Table 3 shows benchmark applications used in our simulations.

7.3 Overall Performance Overheads

This section highlights Penny's low performance overhead compared to prior works. We only show the fault-free execution time overhead since the low soft error rate renders the impact of the recovery procedure on total execution time negligible (see Section 3.2). The following schemes are tested.

- **iGPU** This is De Kruijf et al. [15]'s iGPU [41] that uses anti-dependent register renaming instead of live-out register checkpointing. Note that iGPU requires full ECC-protection for correct recovery.

- **Bolt** This is our GPU adoption of Bolt [35] with the original checkpoint pruning based on a random search. Although most of Penny's optimizations are disabled, Bolt uses our storage alternation to ensure correct recovery without a store buffer. Two versions of Bolt are tested with or without Penny's automatic checkpoint storage assignment.
- **Penny** This is the fully optimized execution of Penny. Checkpoint storages are automatically distributed to shared and global memories by default.

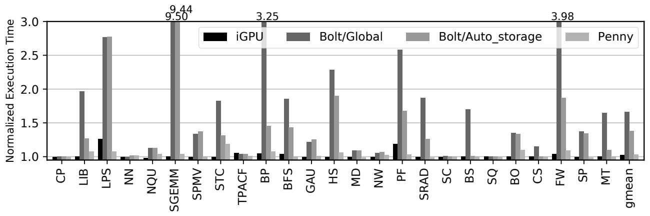


Figure 9. Fault-free execution time overhead.

Figure 9 represents the normalized fault-free execution time overheads of Penny and others compared to the baseline that is the original program with no modification. iGPU shows 2.3% of overhead on average, and up to 26.6%. The slowdown originates from increased register pressure from register renaming, leading to register spills to memory or diminished occupancy. Nevertheless, it would be a mistake to take this to mean that iGPU can be used to replace ECC. Again, unlike Penny, iGPU requires both ECC protection and en(de)coding logic hardening for correct recovery, and therefore such a lower overhead can only be achieved by at the cost of the considerable hardware complexity.

We tested 2 versions of Bolt; Bolt/Global stores all checkpoints to global memory while Bolt/Auto_storage distributes the checkpoints to shared/global memories by using Penny's automatic checkpoint storage assignment. Both versions show significant overhead. That is because unpruned (i.e., committed) checkpointing stores in a loop stall the GPU pipeline significantly. Meanwhile, Bolt/Auto_storage (38.5% overhead) outperforms Bolt/Global (66.5%), which highlights the benefit of Penny's automatic storage assignment.

Finally, Penny reduces Bolt's overhead to 3.3% on average. Most of the applications incur less than 8%; the only exception is STC (19.0%) where loop-carried data-dependences in inner-most loops prevent the checkpoints from being pruned. This is inevitable since the dependences are originated from program semantics that prohibits Penny's checkpoint pruning and bimodal checkpoint placement.

7.4 Impact of Penny's Optimizations

This section investigates the performance impact of Penny's optimizations. To see if they are synergistic, we applied Penny's optimizations one at a time incrementally. That is, each bar of Figure 10 shows the run-time overhead of accumulated optimizations without those in the next bars. For example, the +BCP bar depicts the overhead of applying *bimodal checkpoint placement (BCP)* along with the prior

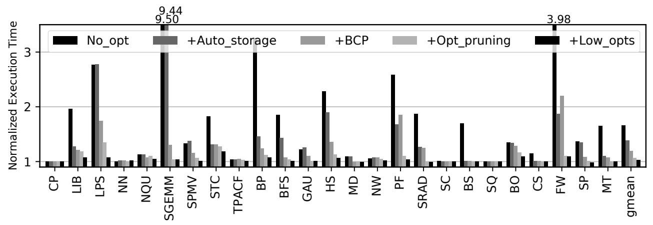


Figure 10. Impact of Penny optimizations accumulated.

automatic storage assignment optimization (ASAO), the overhead of which is represented in the +Auto_storage bar. Similarly, the +Opt_pruning bar depicts the overhead of applying *optimal checkpoint pruning* in combination with prior optimizations (i.e., BCP and ASAO), while the +Low_opts bar shows the overhead of fully-optimized Penny when combining low-level optimizations (Section 6.6) such as LICM with all other prior optimizations. We found out that although individual optimization is sometimes not beneficial by itself, e.g., enabling BCP in PF and FW, its combinations with other optimizations have a synergistic effect. For example, enabling all optimizations (3.3% on average) always outperforms all other combinations of the optimizations.

7.5 Assigning Checkpoint Storage and Its Integrity

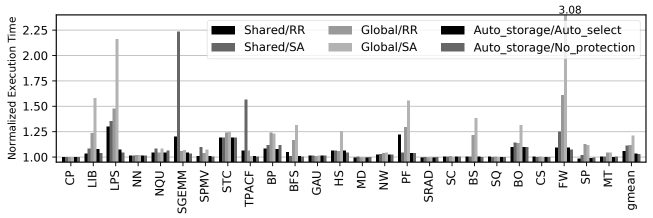


Figure 11. Storage assignment and overwrite prevention.

This section provides sensitivity analysis results on different checkpoint storage assignment schemes and checkpoint overwriting prevention schemes. In Figure 11, the first 4 bars describe the run-time overhead of possible combinations of bimodal storage assignment (Shared/Global) and overwriting prevention, i.e., RR (register renaming) and SA (storage alternation). In the next bar (5th), Auto_storage/Auto_select corresponds to the use of both Penny's automatic storage assignment—that distributes the storages to shared and global memories in a way to maintain the GPU occupancy—and automatic selection of the best between RR and SA. In particular, the 6th bar of the figure shows the overhead of Auto_storage without protecting the checkpoint storage. As shown, the heights of the last 2 bars are almost the same except for LIB and LPS. Thus, Penny's checkpoint overwriting prevention does not incur a noticeable run-time overhead.

7.6 Impact of Optimal Checkpoint Pruning

This section studies the statistics of our optimal checkpoint pruning—that can significantly reduce Penny's run-time overhead, as shown in Section 7.4 in comparison to Bolt's naive pruning. We broke down the total number of checkpoints to 3 parts, the portions of which are described in

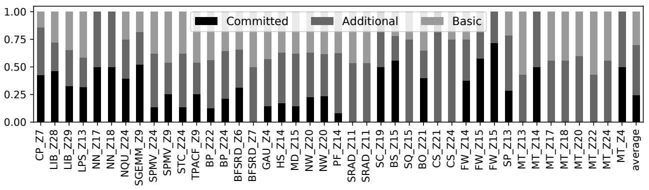


Figure 12. Checkpoints removed by basic/optimal pruning.

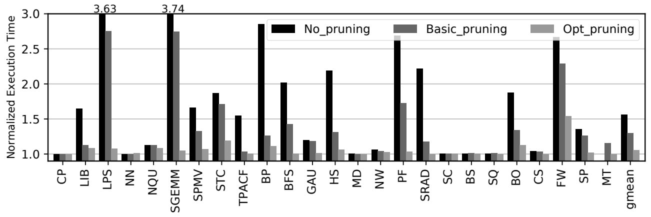


Figure 13. Performance impact of basic/optimal pruning.

Figure 12: (1) Basic corresponds to the checkpoints eliminated by Bolt's basic pruning while (2) Additional to those checkpoints that can further be eliminated only by Penny's optimal pruning. Finally, (3) Committed is the remaining checkpoints after Penny performs the optimal pruning. On average, basic and optimal pruning schemes eliminate the total number of checkpoints by 30% and 75%, respectively.

The eliminated checkpoints translate to the run-time overhead reduction. As shown in Figure 13, when no pruning is enabled, the average overhead becomes 56.2% with a 3.8x slowdown in the worst case. Bolt's basic pruning reduces the overhead down to 29.5%. However, applications like LPS, SGEMM, STC, PF, and FW still cause a large slowdown (up to 274.3% overhead). In contrast, Penny's optimal pruning can handle the applications by removing a checkpoint in their loops, achieving a 5.7% run-time overhead on average.

7.7 Energy Impact on a Register File

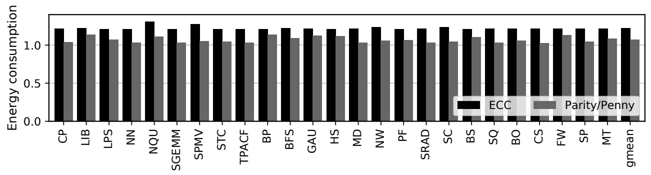


Figure 14. Energy consumption of RF.

In addition to the hardware synthesis (Section 7.1), we evaluated Penny's RF energy benefit over SECDED-ECC using simulation. To measure the actual energy savings on RF for the single-bit error protection, we applied the synthesis data in Table 2 to GPGPU-Sim's power simulator, i.e., GPUWatch[29]. Figure 14 shows the resulting RF energy consumption for each benchmark. It turns out that Penny only consumes 7.0% more energy compared to the baseline RF that has no protection, while the SECDED-ECC RF consumes 22.4% more energy. Additional discussions on the total GPU energy consumption are deferred to Section 9.1.

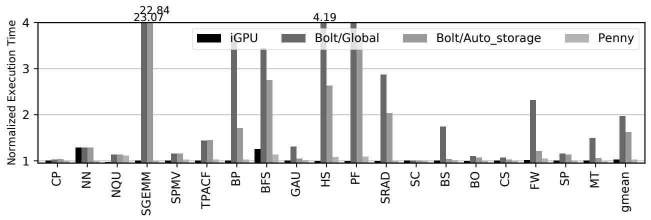


Figure 15. Performance comparison on Titan V.

7.8 Performance Overhead on Volta Architecture

For an architecture sensitivity analysis, this section provides additional simulation results of running Penny on the modern Volta [1] architecture based Titan V GPU. For this purpose, we used an experimental version of GPGPU-sim. However, due to the version incompatibility of CUDA SDK required for the new architecture, we were not able to run a few applications on the GPGPU-sim. Figure 15 shows the fault-free execution time overheads of iGPU, Bolt, and Penny. Although Volta architecture is equipped with much larger caches, it shows almost the same trend observed in the results of old architecture (see Figure 9). Overall, the run-time overhead of Penny is only 3.6% on average.

8 Other Related Work

Recently, researchers have leveraged idempotence for recovery from soft errors [15, 21]. Also, Liu et al. [35] advanced the state of the art with *checkpoint pruning*, which serves to remove checkpoint operations that can be reconstructed from other checkpoints in the event of a soft error. Liu et al. [34, 36, 37] also extend the original idempotent processing in the context of sensor-based soft error detectors to ensure complete recovery. More recently, the energy-harvesting systems [7, 8] have started using idempotent processing to recover from the frequent power failures that occur in systems without batteries [33, 58, 61]. Significantly, all of these projects target CPUs, where store buffers exist.

For GPUs, error resilience studies have focused on systematically evaluating and understanding the impact of errors in GPGPU applications [19, 20, 31, 45]. The most closely-related work is iGPU that leverages idempotent recovery for exception handling, context switching, and timing speculation [41]. However, since iGPU requires the ECC-protected registers and their hardened en(de)coding logic to ensure correct recovery, it cannot be used for achieving ECC-free register file (RF) protection in GPUs. Despite this wealth of related work, Penny is, to the best of our knowledge, the first system to use idempotence to achieve lightweight RF protection without the cost of full ECC-protection.

9 Conclusion

We presented Penny, a compiler-directed resilience scheme for protecting GPU register files against soft errors. To avoid the hardware cost of conventional ECC protection, Penny uses cheaper error detection code (EDC) and idempotent recovery. Penny guarantees correct recovery by preventing checkpoints from being overwritten and significantly reduces their overhead by removing many of them without compromising the recoverability. Across 25 benchmarks, Penny only causes ≈3% run-time overhead on average. The upshot is that Penny allows GPU architects to design their register file (RF) without the ECC cost for equal resilience or achieve stronger resilience using the same ECC cost.

9.1 Limitation and Future Work

Since RF's portion in the total GPU energy consumption might not be dominant, Penny could increase the total energy consumption. Thus, we save the claim on Penny's benefits of the total energy reduction for our future work that will conduct more design space exploration and performance optimization to fully realize the benefits. Apart from that, it is still critical to reduce the RF energy itself. The reason is that a register file (RF) determines the GPU's nominal voltage (Vdd) that must be set high enough to handle the worse-case voltage demand [30]. In fact, RF's burst accesses originated by GPU's massive parallelism often cause large voltage swings in the power delivery, which must be guarded by sufficiently-high Vdd. If Penny is used to reduce the RF energy, GPU architects can lower the operating voltage, thereby improving the entire GPU's energy-efficiency.

A Correctness of Penny's Recovery

This section proves that when EDC (parity) based detection is combined with Penny's idempotent recovery, RF error can be safely recovered even without enforcing the in-region detection required for all prior idempotent recovery schemes.

A.1 Prevention of Error Propagation

We first show that when EDC is used to detect errors in RF, they are never propagated to any other location (register/memory) before their register corruption is first detected.

Axiom 1. Given instruction execution, if register is corrupted, parity error is detected at the moment of the register access.

Following two theorems are to prove the impossibility of error propagation for a single error and multiple errors, respectively, in the presence of parity checking.

Theorem A.1. If register r is corrupted and then detected at a point P for the first time, the corrupted value has not yet been propagated to other locations before P.

Proof. We use proof by contradiction. Suppose the argument is false, meaning that the corruption had been propagated since some point before P. For r's corrupted value to be propagated, r must be first read as a source operand of an instruction. At the point of the instruction execution, r's corruption must be detected by its parity checking (Axiom 1). This contradicts the fact that P is the first point to recognize that r is corrupted. □

Theorem A.2. If r's corruption is detected at a point P for the first time and other corrupted registers have not been detected before P, then they have not been propagated to other locations.

Proof. We use proof by contradiction. Suppose the argument is false, e.g., some other corrupted register $r2$ had been propagated since some point before P. For $r2$'s corrupted value to be propagated, $r2$ must be first read in which case the corruption must be detected momentarily (Axiom 1). This is another contradiction from the premise that P is the first point to detect r's corruption. □

The lack of error propagation implies that at the point of the parity error detection in a region R, we can trust all register values saved in Penny's checkpoint storages that are protected by ECC in GPU cache/memory.

A.2 Proof of Correct State Recovery

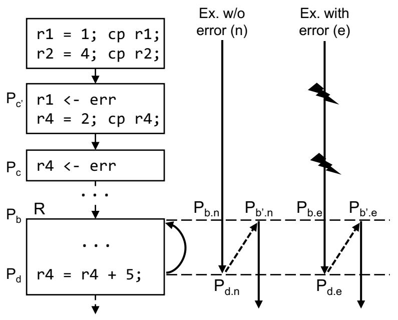
This section shows that Penny correctly recovers the required memory and RF state—even in the presence of multiple corrupted registers. Let's define Val, Reg, and Loc as a set of values, registers, and memory locations, respectively. To describe program execution states at a given program point P, we use a 3-tuple $\langle RF(P), MEM(P), CP(P)\rangle$ where $RF(P) : Reg \rightarrow Val$ corresponds to the state of the register file while $MEM(P) : Loc \rightarrow Val$ to the memory state excluding the checkpoint storage state that is described by $CP(P) : Reg \rightarrow Val$.

We introduce a few functions to be used in our proof: $MEM(P_1)|_{live(R)}$ and $RF(P_1)|_{live(R)}$ signify the subset of memory and register states (values) at a program point P_1 which consists of only the locations and registers live at the beginning of a region R. Similarly, $CP(P)|_{livein(R)}$ gives the subset of checkpoint storages at a point P which consists of only the live-in registers of a region R. Also, $RF(P)[CP(P)|_{livein(R)}]$ represents updating the register file state $RF(P)$ with the checkpointed values of R's live-in registers at a point P, i.e., restoring input registers of the region using its checkpointed live-in registers for recovery.

At the core of our proof, we compare two execution scenarios shown as n and e in Figure 16—normal execution with no error (n) and errant one (e) where errors can be detected and corrected by Penny—and show both executions result in the same program execution states.

For errant execution (e), an error occurred in P_c and it is detected at P_d within region R—the 2 points can be far apart separated by multiple regions while undetected errors could exist (e.g., $P_{c'}$) if they have not been read yet. P_b depicts the entry point of the region R; we also use $P_{b'}$ to represent the re-execution of the entry after the error detection.

For normal execution (n), at P_d, we trigger the re-execution of the region R which is preceded by the restoration of live-in



```
r1 = 1; cp r1;
r2 = 4; cp r2;

r1 <- err
r4 = 2; cp r4;

r4 <- err

...

r4 = r4 + 5;
```

Figure 16. Safely recovering from errors across regions

registers, for comparison to errant execution (e). To differentiate program execution states between the 2 executions, their program points use 2 suffixes $.e$ and $.n$, respectively.

To show both executions (n and e) generate the same program state, i.e., $\langle RF(P), MEM(P), CP(P)\rangle$, we first prove that live register values at R's entry in n are identical to those in e when Penny restarts R.

Lemma A.3. Live register values at P_b in normal execution n are the same as the restored register values at $P_{b'}$, i.e., when the region R is re-executed for error recovery.

Proof.

$$RF(P_{b.n})|_{live(R)} = RF(P_{d.n})[CP(P_{d.n})|_{livein(R)}]|_{live(R)} \tag{1}$$

$$RF(P_{d.n})[CP(P_{d.n})|_{livein(R)}]|_{live(R)} = RF(P_{d.e})[CP(P_{d.e})|_{livein(R)}]|_{live(R)} \tag{2}$$

$$RF(P_{d.e})[CP(P_{d.e})|_{livein(R)}]|_{live(R)} = RF(P_{b'.e})|_{live(R)} \tag{3}$$

Equation 1 implies that live register values at the region entry point $P_{b.n}$ can be safely restored at $P_{d.n}$ by loading the checkpointed values corresponding to live-in registers of R. This must be true because of 2 reasons: (1) Penny's checkpoint scheduling ensures that all live-out registers of a region are checkpointed before the region ends, thus all live register values at P_b have already been checkpointed before entering the region R, and (2) Penny's overwriting prevention technique preserves the checkpointed register values until the end of the region R. Equation 2 states that although registers are corrupted in errant execution (e), the restored live register values must be the same as those in normal execution (n). This is true because corrupted register values can never be propagated to anywhere else, thus checkpoint storages remain intact (Theorem A.1, A.2). Lastly, Equation 3 tells that these restored register values are used in R's re-execution for error recovery. This is true by the definition of idempotent recovery (Section 3.1). □

Now we prove that memory values are identical in n, e.

Lemma A.4. Live memory values at P_b in normal execution n are the same as those at $P_{b'}$, i.e., when the region R is re-executed for error recovery.

Proof.

$$MEM(P_{b.n})|_{live(R)} = MEM(P_{d.n})|_{live(R)} \tag{4}$$

$$MEM(P_{d.n})|_{live(R)} = MEM(P_{d.e})|_{live(R)} \tag{5}$$

$$MEM(P_{d.e})|_{live(R)} = MEM(P_{b'.e})|_{live(R)} \tag{6}$$

Equation 4 states that in normal execution n, live memory values at region entry $P_{b.n}$ are not overwritten at $P_{d.n}$, which is true because idempotent region formation ensures no memory anti-dependences in each region. Equation 5 then tells that despite the errors, live memory values at P_d in errant execution e is the same as those in normal execution. This must be true because, due to the error propagation prevention of parity checking (Theorem A.1, A.2), all memory values remain intact, i.e., $MEM(P_{d.n}) = MEM(P_{d.e})$, regardless of errors. Finally, Equation 6, i.e., the live memory values remain the same between the error detection and R's re-execution, must be true since Penny's recovery block never updates memory. □

Finally, we prove checkpoint storages are identical in n, e.

Lemma A.5. Checkpointed values of R's live-in registers at P_b in normal execution n are the same as those at $P_{b'}$, i.e., when the region R is re-executed for error recovery.

Proof. Penny's checkpoint overwriting prevention ensures that $CP(P_{b.n})|_{livein(R)}$ should remain the same during R's execution. Due to Theorem A.1, A.2, an error cannot change any of checkpointed values. In addition, since Penny's recovery block on an error does not change them, it is true that $CP(P_{b.n})|_{livein(R)} = CP(P_{b'.e})|_{livein(R)}$. □

We have proven that all live memory/register/checkpoint states of errant execution (e) upon recovery are equivalent to those of normal execution (n). Consequently, Penny's recovery is correct though it does not enforce the in-region detection. Note that other undetected errors in RF, e.g., one at P'_c, are spontaneously corrected at the same recovery time at which all live-in register values are restored by loading their checkpointed values. Corruptions in non-live-in registers may remain but do not affect program correctness because they will never be read before being written.

Acknowledgment

We appreciate PLDI reviewers and our shepherd Sriram Krishnamoorthy for their constructive comments. At Purdue University, this work was supported by NSF grants 1750503 (CAREER) and 1814430. At Seoul National University, this work was supported in part by the National Research Foundation of Korea (NRF) grants (No. NRF-2016M3C4A7952587 and No. NRF-2019M3E4A1080386), by the BK21 Plus program for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU, No. 21A20151113068) through NRF, and by the Institute for Information & communications Technology Promotion (IITP) grant (No. 2018-0-00581, CUDA Programming Environment for FPGA Clusters), all funded by the Ministry of Science and ICT (MSIT).

References

[1] 2017. *NVIDIA Tesla V100 GPU Architecture.* Technical Report. Nvidia.

[2] Gene M Amdahl. 2013. Computer architecture and amdahl's law. *Computer* 46, 12 (2013), 38–46.

[3] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS'09.*

[4] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H Albonesi. 2001. Reducing the complexity of the register file in dynamic superscalar processors. In *MICRO 34.*

[5] Shekhar Borkar. 2013. Exascale Computer-a fact or a fiction. *Keynote address: IEEE International Parallel and Distributed Processing Symposium* (2013).

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC 2009.*

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

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

[9] Cristian Constantinescu. 2003. Trends and Challenges in VLSI Circuit Reliability. In *MICRO 36.*

[10] W.J. Cook, W.H. Cunningham, W.R. Pulleyblank, and A. Schrijver. 2011. *Combinatorial Optimization.* Wiley.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[12] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. 2010. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *ISCA'10.*

[13] Marc de Kruijf and Karthikeyan Sankaralingam. 2013. Idempotent code generation: Implementation, analysis, and evaluation.. In *CGO'13.*

[14] Marc A. De Kruijf. 2012. *Compiler Construction of Idempotent Regions and Applications in Architecture Design.* Ph.D. Dissertation. Madison, WI, USA. Advisor(s) Sankaralingam, Karthikeyan.

[15] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static Analysis and Compiler Design for Idempotent Processing. In *PLDI 2012.*

[16] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The Transmeta Code Morphing&Trade; Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (San Francisco, California, USA). 15–24.

[17] Jeno Egerváry. 1931. Matrixok kombinatorius tulajdonságairól. *Matematikai és Fizikai Lapok* 38, 1931 (1931), 16–28.

[18] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *ISCA'11.*

[19] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. 2014. GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications. In *ISPASS'14.* IEEE.

[20] Bo Fang, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. 2016. A systematic methodology for evaluating the error resilience of GPGPU applications. *IEEE Transactions on Parallel and Distributed Systems* 27, 12 (2016), 3397–3411.

[21] Shuguang Feng, Shantanu Gupta, Amin Ansari, Scott A Mahlke, and David I August. 2011. Encore: low-cost, fine-grained transient fault recovery. In *MICRO 44.*

[22] L Bautista Gomez, Franck Cappello, Luigi Carro, Nathan DeBardeleben, Bo Fang, Sudhanva Gurumurthi, Karthik Pattabiraman, Paolo Rech, and M Sonza Reorda. 2014. GPGPUs: how to combine high computational power with high reliability. In *Proceedings of the conference on Design, Automation and Test in Europe.* 341.

[23] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2011. Toward Dark Silicon in Servers. In *MICRO 31.*

[24] Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. 2013. Reliable On-chip Systems in the Nano-era: Lessons Learnt and Future Trends. In *DAC 2013.*

[25] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. 2012. Near-Threshold Voltage (NTV) Design: Opportunities and Challenges. In *DAC'12.*

[26] Seon Wook Kim, Chong-Liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. 2006. Exploiting Reference Idempotency to Reduce Speculative Storage Overflow. In *TOPLAS'06.*

[27] Denés Konig. 1931. Gráfok és mátrixok. Matematikai és Fizikai Lapok.

[28] Lingbo Kou. 2014. *Impact of Process Variations on Soft Error Sensitivity of 32-nm VLSI Circuits in Near-threshold Region.* Master's thesis.

[29] Jingwen Leng, Tayler H. Hetherington, Ahmed ElTantawy, Syed Zohaib Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: enabling energy optimizations in GPGPUs. In *ISCA'13.*

[30] Jingwen Leng, Yazhou Zu, and Vijay Janapa Reddi. 2015. GPU voltage noise: Characterization and hierarchical smoothing of spatial and temporal voltage noise interference in GPU architectures. In *HPCA'15.*

[31] Guanpeng Li, Karthik Pattabiraman, Chen-Yong Cher, and Pradip Bose. 2016. Understanding Error Propagation in GPGPU Applications. In *SC'16.*

[32] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. 2018. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *MICRO 51.*

[33] Qingrui Liu and Changhee Jung. 2016. Lightweight Hardware Support for Transparent Consistency-Aware Checkpointing in Intermittent Energy-Harvesting systems. In *NVMSA 2016.*

[34] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2015. Clover: Compiler Directed Lightweight Soft Error Resilience. In *LCTES'15* (Portland, OR, USA).

[35] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Compiler-Directed Lightweight Checkpointing for Fine-Grained Guaranteed Soft Error Recovery. In *SC'16.*

[36] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Compiler Directed Soft Error Detection and Recovery to Avoid DUE and SDC via Tail-DMR. *TECS'16* (2016).

[37] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Low-Cost Soft Error Resilience with Unified Data Verification and Fine-Grained Recovery. In *MICRO 49.*

[38] Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. 1992. Sentinel Scheduling for VLIW and Superscalar Processors. In *ASPLOS V.*

[39] Gokhan Memik, Masud H Chowdhury, Arindam Mallik, and Yehea I Ismail. 2005. Engineering over-clocking: Reliability-performance tradeoffs for high-performance register files. In *DSN'05.*

[40] Gokhan Memik, Mahmut T. Kandemir, and Ozcan Ozturk. 2005. Increasing Register File Immunity to Transient Errors. In *DATE.*

[41] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. 2012. iGPU: Exception Support and Speculative Execution on GPUs. In *ISCA'12.*

[42] Pablo Montesinos, Wei Liu, and Josep Torrellas. 2007. Using register lifetime predictions to protect register files against soft errors. In *DSN'07.*

[43] Todd K Moon. 2005. *Error correction coding: mathematical methods and algorithms.* John Wiley & Sons.

[44] S.S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers.

[45] Bin Nie, Lishan Yang, Adwait Jog, and Evgenia Smirni. 2018. Fault Site Pruning for Practical Reliability Analysis of GPGPU Applications. In *MICRO 51*. IEEE.

[46] Nvidia 2007. *CUDA Programming Guide*. Nvidia. http://developer.download.nvidia.com/compute/cuda.

[47] Nvidia 2013. *CUDA Toolkit 5.5*. Nvidia. https://developer.nvidia.com/cuda-toolkit-55-archive.

[48] Robert Pawlowski. 2015. *Measurement and Analysis of Soft Error Vulnerability of Low-Voltage Logic and Memory Circuits*. Ph.D. Dissertation. Corvallis, OR, USA.

[49] William Wesley Peterson and EJ Weldon. 1972. *Error-correcting codes*. MIT press.

[50] George A Reis, Jonathan Chang, Neil Vachharajani, Shubhendu S Mukherjee, Ram Rangan, and David I August. 2005. Design and evaluation of hybrid fault-detection systems. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 148–159.

[51] Muhammad Shafique, Siddharth Garg, Jörg Henkel, and Diana Marculescu. 2014. The EDA Challenges in the Dark Silicon Era: Temperature, Reliability, and Variability Perspectives. In *DAC '14*.

[52] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Technical Report IMPACT-12-01* 127 (2012).

[53] Synopsys. 2001. Compiler, Design and User, RTL and Guide, Modeling. (2001). http://www. synopsys. com.

[54] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[55] Michael B. Taylor. 2012. Is Dark Silicon Useful?: Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *DAC'12* (San Francisco, California).

[56] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, L. Carro, and A. Bland. 2015. Understanding GPU errors on large-scale HPC systems and the implications for system design and operation. In *HPCA '15*. IEEE.

[57] Marc Tremblay and Yu Tamir. 1989. Support for fault tolerance in VLSI processors. In *Circuits and Systems, 1989., IEEE International Symposium on*. IEEE, 388–392.

[58] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent computation without hardware support or programmer intervention. In *OSDI'16*.

[59] Liang Wang and Kevin Skadron. 2013. Implications of the Power Wall: Dim Cores and Reconfigurable Logic. *IEEE Micro* (2013), 40–48.

[60] S.J.E. Wilton et al. 1996. CACTI: An enhanced cache access and cycle time model. *JSSC'96* (May 1996).

[61] Mimi Xie, Mengying Zhao, Chao Pan, Jingtong Hu, Yongpan Liu, and Chun Xue. 2015. Fixing the Broken Time Machine: Consistency-Aware Checkpointing for Energy Harvesting Powered Non-Volatile Processor. In *DAC'15*.

[62] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. 2018. CRAT: Enabling Coordinated Register Allocation and Thread-Level Parallelism Optimization for GPUs. *TC'08* (2018).

[63] Doe Hyun Yoon and Mattan Erez. 2009. Memory Mapped ECC: Low-cost Error Protection for Last Level Caches. In *ISCA'09* (Austin, TX, USA).