Doctoral Thesis

Compiler Driven Soft Error Protection Techniques for GPUs

GPU 에러 안정성 보장을 위한 컴파일러 기법

BY

Hongjune Kim

August 2020

Graduate School of Seoul National University
Department of Electrical Engineering & Computer Science
College of Engineering

Doctoral Thesis

Compiler Driven Soft Error Protection Techniques for GPUs

GPU 에러 안정성 보장을 위한 컴파일러 기법

BY

Hongjune Kim

August 2020

Graduate School of Seoul National University
Department of Electrical Engineering & Computer Science
College of Engineering

Compiler Driven Soft Error Protection Techniques for GPUs

GPU 에러 안정성 보장을 위한 컴파일러 기법

Adviser Jaejin Lee

Submitting a doctoral thesis of Electrical Engineering & Computer Science

August 2020

Graduate School of Seoul National University

Department of Electrical Engineering & Computer Science

Hongjune Kim

Confirming the doctoral thesis written by Hongjune Kim

August 2020

Chair : ____________________________

Vice Chair: ____________________________

Examiner : ____________________________

Examiner : ____________________________

Examiner : ____________________________

Abstract

Due to semiconductor technology scaling and near-threshold voltage computing, soft error resilience has become more important. Nowadays, GPUs are widely used in high performance computing (HPC) because of its efficient parallel processing and modern GPUs designed for HPC use error correction code (ECC) to protect their storage including register files. However, adopting ECC in the register file imposes high area and energy overhead.

To replace the expensive hardware cost of ECC, we propose Penny, a lightweight compiler-directed resilience scheme for GPU register file protection. We combine recent advances in idempotent recovery with low-cost error detection code. Our approach focuses on solving two important problems:

1. *Can we guarantee correct error recovery using idempotent execution with error detection code?* We show that when an error detection code is used with idempotence recovery, certain restrictions required by previous idempotent recovery schemes are no longer needed. We also propose a software-based scheme to prevent the checkpoint value from being overwritten before the end of the region where the value is required for correct recovery.

2. *How do we reduce the execution overhead caused by checkpointing?* In GPUs additional checkpointing store instructions inflicts considerably higher overhead compared to CPUs, due to its architectural characteristics, such as lack of store buffers. We propose a number of compiler optimizations techniques that significantly reduce the overhead.

keywords: GPU, Resilience, ECC, Idempotence

student number: 2012-30204

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Why is Soft Error Resilience Important in GPUs?

Due to technology scaling and near-threshold computing [22, 32, 39, 77, 85], 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 makes it harder to predict the response of the circuits to a particle strike, thus making them more susceptible to soft errors [17, 32, 39, 40, 41, 45, 68, 72, 77, 85].

HPC applications are particularly vulnerable to the outcome of undesirable soft errors, *e.g.*, SDC, because of their long-running nature on the large-scale systems, such as supercomputers and datacenters, In fact, soft error resilience is one of the key Exascale research challenges [56, 16, 80, 7, 5, 4, 73].

With the popularity of GPUs, it is becoming more important to protect them against soft errors [36, 78]. 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 [12, 58, 59, 81], e.g., taking up to 3 times the delay of ALU operations [58, 59, 81]. Due to the increased delay and power [61], ECC-protected RF consumes significantly more energy than ECC-free RF. More importantly, its energy consumption may become substantially larger than that of a register access [8, 59, 58] due to the increased delay and power [61].

Indeed, the RF experiences the largest and fastest current changes in a GPU [48]. Thus, RF accesses are often the root cause of large voltage droops [47, 48]. This implies that ECC-free RFs can achieve a significant voltage guardband reduction [10, 9, 13], thereby improving the GPU energy efficiency.

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. [1]. Since they cannot be handled by conventional single-bit error correction and double-bit error detection (SECDED) ECC [89], 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.

Table 1.1 presents the trend of the continuously increasing register file size in GPUs as the microarchitecture evolves over generations.

Even if it is effective in decreasing undesirable soft error outcomes such as SDC, this comes with a significant performance penalty. Disabling the ECC protection of NVIDIA Tesla M2090 reduces the 10-hour-long simulation time of molecular dynamics simulation to 9 hours [84, 15]. For this reason, when performing computationally intensive calculations, the ECC is usually turned off to avoid the associated penalty [30,

[1]Future generations of GPUs are likely to face with the high demand for resilience against multi-bit errors due to aggressive technology scaling (e.g., 7nm AMD Vega GPU) and near-threshold voltage (NTV) operation which is known to increase multi-bit errors by 2.6x [68]

	uArchi- tecture	Per SM RF size (KB)	Number of SM	Total RF size (KB)
GT200	Tesla	64	30	1920
GF100	Fermi	128	16	2048
GK110	Kepler	256	15	3840
GM200	Maxwell	256	24	6144
GP100	Pascal	256	56	14336

Table 1.1: The trend of GPU register file size increase across microarchitectures in different generations; given the ever growing trend, protecting RFs with ECC dramatically increases both the hardware complexity and the power consumption.

66].

Given all this, there is a compelling need for lightweight GPU RF protection.

1.2 How can the ECC Overhead be Reduced?

With that in mind, we propose Penny, a new GPU RF resilience scheme that combines recent advances in idempotent recovery [27, 28, 35, 50, 52, 53, 54, 60] 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 [69] 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 [28]. Thus, the program can recover from soft errors by simply restarting the idempotent region where they occurred. Among the existing schemes, Bolt [53] 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 [50, 25, 27, 29, 35, 52, 54, 60]. To achieve correct soft error recovery without ECC, Bolt checkpoints the live-out registers of idempotent regions.

1.3 What are the Challenges?

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 [27, 29, 35, 52, 53, 54, 60]. 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 [67], shows a 26.7% slowdown when only 2 checkpointing stores are added into the inner-most loop.

1.4 How do We Solve the Challenges?

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.

The major contribution of this work is also published in a conference paper [42].

Chapter 2

Comparison of Error Detection and Correction Coding Schemes for Register File Protection

GPUs are equipped with a large and highly banked RF—that can hold the contexts of concurrently running thousands of threads in active warps—to enable fast hardware context switching between the warps, which is a key for GPU's long latency hiding technique. For example, recent generations of Nvdia architecture comprises 256 KB registers per SM, adding up to 20 MB in total for Tesla V100 GPU [1] and 18 MB for Quadro RTX 6000 [2]. To recover from RF errors, GPUs for HPC have equipped RF with ECC protection.

The conventional way to protect register files from soft errors is to use error correcting codes (ECCs) and modern GPUs used for high performance computing commonly adopt ECC in data storages including the register file. However, since register file size in GPUs is huge, RF ECC occupies large die area and often lengthens the clock cycle [12, 58, 59, 81]. Due to increased area, delay, and access power [61], ECC-protected RF consumes significantly more energy than ECC-free RF.

Coding scheme	Data bits	Additional bits	Encoded bits	Overhead	Detectable error bits	Recoverable error bits
Parity	32	1	33	3.1%	1	-
Hamming	32	6	38	18.8%	2	-
SECDED	32	7	39	21.9%	2	1
DECTED	32	23	55	71.9%	3	2
TECQED	32	28	60	87.5%	4	3

Table 2.1: Coding schemes used for 32-bit error detection and correction.

2.1 Error Correction Codes and Error Detection Codes

Error correction codes (ECC) require to correct corrupted bits in data while error detecting codes (EDC) only needs to detect corruptions, so EDC requires more redundant information in the extra parity bits.

Table 2.1 compares the capability of coding schemes used for ECC and EDC to protect 32-bit of data. For 32-bit of original *data bits*, *additional bits* are added and encoded into total size presented in *encoded bits*. *Overhead* is the size overhead of extra bits. *Detectable error bits* are the number of error bits properly detected and *recoverable error bits* are the number of error bits that can be safely recovered from. For example, when a single bit parity is used, a single bit of error data corruption can be detected but they cannot be corrected. This is called detected but unrecoverable errors (DUEs). To detect up to two bits of data corruption Hamming (38, 32) code [1] can be used.

Single error correction double error detection (SECDED) [62] is a widely used ECC coding scheme. When there is single-bit data corruption, SECDED can safely detect it and correct it using the parity data. It can also detect double-bit errors, but it cannot be corrected resulting in DUE. When there is a triple bit error in the data, SECDED parity checking can detect it as an error, but it cannot be distinguished from a single bit error, so it is mis-corrected resulting in a faulty execution. For two or three bits of

[1]In coding theory, (n, k) generally stands for encoding k-bits of original data with n-bits [62].

error recovery, ECC must use double error correction triple error detection (DECTED) and triple error correction quadruple error detection (TECQED) respectfully, requiring much higher bit redundancy. For TECQED, we assume a 2-dimensional encoding scheme [3] that splits 32-bit data into 8-bit rows.

2.2 Cost of Coding Schemes

Recoverable error bits	Coding scheme	Area (mm^2)	Access latency (ns)	Access energy (pJ)	Leakage power (nW)
None	Base RF	0.105	1.01	9.64	4.57
1 bit	SECDED (39,32)	0.150 (21.9%)	1.27 (25.6%)	11.68 (21.1%)	5.51 (20.7%)
2 bits	DECTED (55,32)	0.196 (40.6%)	1.50 (49.2%)	13.43 (39.2%)	6.32 (38.4%)
3 bits	TECQED (60,32)	0.335 (87.5%)	1.76 (74.3%)	17.79 (84.5%)	8.35 (82.7%)

Table 2.2: Cost of using conventional ECC for error protection (22nm, per bank).

Recoverable error bits	Coding scheme	Area (mm^2)	Access latency (ns)	Access energy (pJ)	Leakage power (nW)
None	Base RF	0.105	1.01	9.64	4.57
1 bit	Parity (33,32)	0.111 (3.1%)	1.04 (3.5%)	9.93 (3.0%)	4.70 (3.0%)
2 bits	Hamming (38,32)	0.143 (18.8%)	1.23 (21.8%)	11.39 (18.1%)	5.38 (17.7%)
3 bits	SECDED (39,32)	0.150 (21.9%)	1.27 (25.6%)	11.68 (21.1%)	5.51 (20.7%)

Table 2.3: Cost of using EDC in Penny (22nm, per bank).

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 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[2] 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

[2]Parity-check is a general term used to signify the process of validating the encoded data, regardless of the used coding scheme [62].

the same bit-redundancy budget, the number of error-bits ECC can detect is smaller than what EDC can do.

To protect RF using ECC, each 32-bit register can be encoded in SECDED (39, 32), DECTED (55, 32), and TECQED (39, 32) to guarantee safe recovery from 1-bit, 2-bit, and 3-bit of register corruption. Again, notation (n, k) in a encoding scheme means n-bits are required for encoding k-bits of data [62]. In our approach, EDC only has to detect register corruption and the following idempotent recovery can safely restore correct values. Thus, we can safely use single bit parity (33, 32), Hamming (38, 32), and SECDED (39, 32) for 1 to 3 bits of error detection.

To estimate the cost of coding schemes to protect register file, we designed each of them using 22nm in CACTI 6.5 [86]. To model the encoder/decoder overhead, we implemented the designs based on the specifications in Lattice [71] where the 256KB RF is divided into 16 banks. We also used Synopsys design compiler [75] to synthesize the built designs for their evaluation.

Table 2.2 and Table 2.2 is the result of our modeling each coding schemes required for using ECC and EDC in Penny. The first row on each table shows a base RF setting with no encoding scheme and the following rows show the result of coding schemes required to recover from 1 to 3 bits of error. Ratios in the parenthesis show the additional overhead of each value compared to the base RF.

For recover from a single-bit error, SECDED ECC requires 21.9% of additional die area cost, while the single-bit parity EDC used in Penny only adds 3.1% more. 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[3], 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) [62] 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 and correct them by re-executing the idempotent

[3]Multi-bit errors increase by 2.6X under near-threshold operations [68].

region where they occurred. For 3-bit error correction, ECC must use TECQED (60,32) coding that requires 28 additional bits, while Penny can use SECDED (39,32) coding paying only 7 bits to achieve the same correction.

Since SECDED has a Hamming distance of 4 [62], we can re-use it as EDC to detect three-bits of errors. In SECDED ECC, this could not be utilized because three-bit errors cannot be distinguished from a single bit error, so they are mis-corrected as a single-bit error leading to faulty execution. So only single-bit errors (corrected) and two-bit errors (DUE) are handled and more than three-bit errors are assumed not to happen.

We observe that access latency/energy and leakage power show similar trends as the area. Note that in ECC, the overhead costs grow rapidly for protecting from wider cardinality of multi-bit errors, Likewise, the cost difference between ECC and EDC gets much larger.

By utilizing the lightweight EDC in Penny, the system designer can select a proper level of encoding scheme to balance between two goals that are in a tradeoff relation: reducing the hardware cost or providing higher resilience. For example, if only single-bit error resilience is required, SECDED ECC with 21.9% of area overhead can be replaced with single-bit parity with much reduced 3.1% overhead for the same level of resilience. Alternatively, by paying the same hardware cost, Penny can guarantee much stronger resilience, i.e. correct from wider multi-bit errors. In conventional ECC, SECDED can only safely recover from 1-bit errors, but Penny can use the same coding scheme to safely recover from 3-bits of errors.

2.3 Soft Error Frequency of GPUs

To select the right coding scheme for error protection it is important to estimate how frequently soft error occurs on a GPU. Oak Ridge National Laboratory has reported the error characteristics of the Titan supercomputer [79], which consists of 18,688 Tesla

K20X GPUs.

The mean time between failure (MTBF) of single-bit errors (SBE) is 9.04 seconds, meaning that 9558 errors occur every day. When divided by the number of GPUs MTBF is 47 hours, which can be translated to approximately 0.5 errors per day.

Double bit errors (DBE) are much rare compared. MTBF for the full system is 160 hours, meaning approximately 1 error occurs every weak. One interesting thing is register file errors account for 14% of the double bit errors, which is particularly large considering the relative size of the register file compared to the device memory. The authors speculate that this is because the register file using a less effective interleaving technique, in order to reduce area and access-latency overhead. This implies that in environments where multi-bit error resilience is required, register file inflicts higher ECC costs compared to other storages and notably more vulnerable to errors, so it is the best candidate that can be replaced with a light-weight compiler-directed scheme.

Chapter 3

Idempotent Recovery and Challenges

3.1 Idempotent Execution

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 [43, 50, 53, 57, 60, 21, 83].

3.2 Previous Idempotent Schemes

Penny is built upon some ideas from a number of previous idempotent processing techniques. De Kruijf *et al.* presented a technique to transform a program into a sequence of natural idempotent regions [29]. But this technique, like other idempotent recovery schemes [52, 54, 27, 35], requires RFs to be protected by ECCs. For soft error recovery, Bolt [53] 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.

3.2.1 De Kruijf's Idempotent Translation

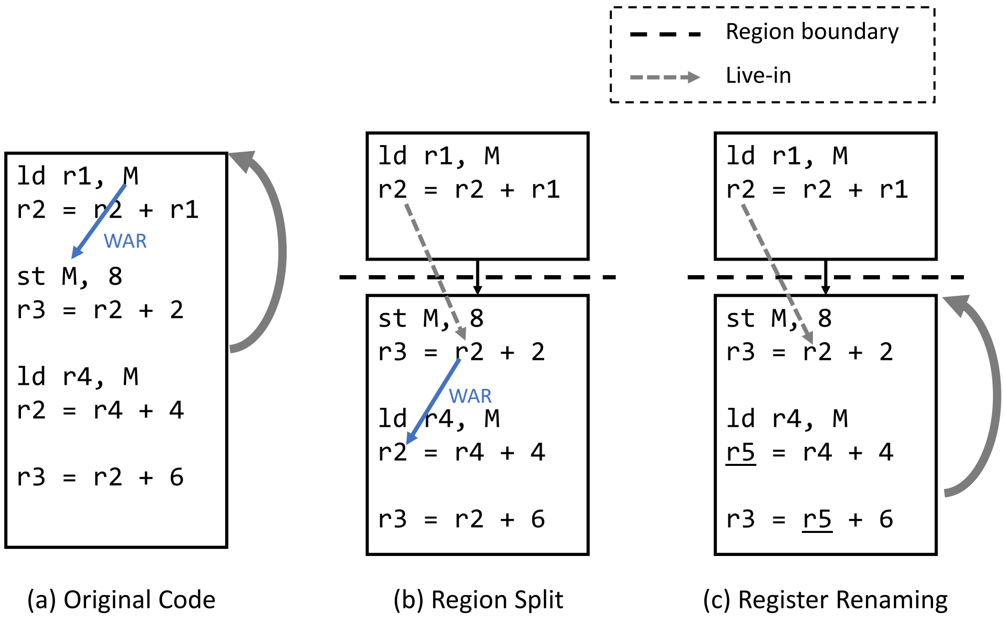


Figure 3.1: Idempotent translation by De Kruijf.

Figure 3.1 shows how De Kruijf's scheme is used to translate a code into idempotent regions. In the original code (a), there is a memory anti-dependence. This prevents from re-executing the code. If the code is restarted as the curved arrow, the overwritten value 8 will be read at the first load, different from the original value.

We cut the anti-dependence into separate regions as in (b), so the memory write will overwrite the loaded value on re-execution. Value of $r2$ from the first region is live-in into the second region, but after the use, there is a write on the register (e.g. register anti-dependence). In such a case, the algorithm renames the overwriting register and its later uses into an unused register. In (c) the overwriting $r2$ and its references are renamed to $r5$. Now, both the first region and the second region can be re-executed from any point in the execution.

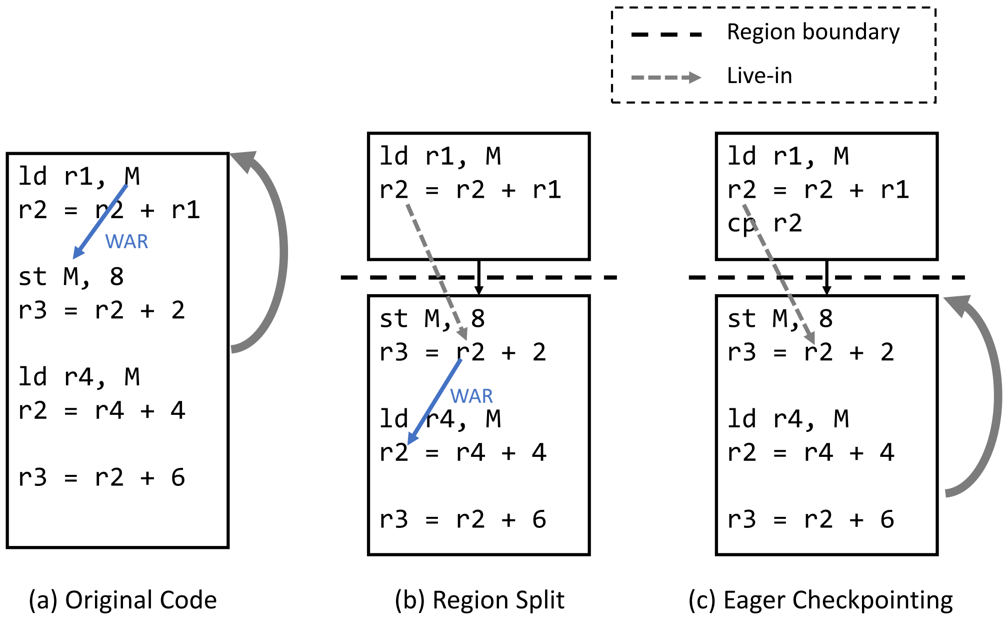


Figure 3.2: Bolt's idempotent translation.

3.2.2 Bolts's Idempotent Recovery

Figure 3.2 is the example of how Bolt translates a code into a safe eager-checkpointing based code. The first step to handle the memory anti-dependence is the same as De Kruijf's algorithm. But for register anti-dependences, the live-in values are checkpointed before entering the region. For example, $r2$ is the live-in value of the second region. The last updated point of the live-in value is found and the value is checkpointed right after the update. This value is stored into an ECC protected storage, so when the re-execution is triggered in the second region, the saved value is read from the checkpoint to restore the live-in register value.

3.2.3 Comparison between Idempotent Schemes

Table 3.1 compares Penny and the idempotent scheme it is closely related to. To handle memory anti-dependences, De Kruijf separates the anti-dependence into separate regions. Bolt and Penny adopt this technique to handle memory anti-dependence. Penny

Idempotence scheme	De Kruijf	Bolt	Penny
Memory Anti-dependence	Heuristic region cut	Heuristic region cut	Heuristic region cut, Cost-aware region cut
Register Anti-dependence	Register renaming	Eager checkpointing	Bimodal checkpoint placement
Checkpoint Over-write Prevention	-	Gated store buffer	Storage alternation, Register renaming
Checkpoint Pruning	-	Brute-force algorithm	Optimal algorithm
Register Corruption Recovery	No	Yes	Yes

Table 3.1: Comparison of idempotent schemes.

also introduces a new checkpoint cost-aware algorithm. While De Kruijf renames the anti-dependences on live-in registers to transform the region idempotent, Bolt and Penny checkpoints the live-out registers. Bolt checkpoints the register right after the last update point, but Penny selectively delays the checkpoint to the end of the region when profitable. While Bolt uses a hardware gated store buffer to delay the checkpoints in order not to overwrite checkpointed values that are live-ins of the current region, Penny introduces two software techniques since GPUs generally do not consist of store buffers.

Most importantly, merely translating code into idempotent regions using De Krujif's algorithm does not provide a way to recover from register corruptions. Some of the following work uses this translation in GPU exception handling and speculation [60], simplifying in-order processor [26], and concurrency bug recovery [90]. On the contrary, Bolt suggests an efficient way to recover from register corruptions using eager checkpointing and idempotent re-execution.

3.3 Idempotent Recovery Process

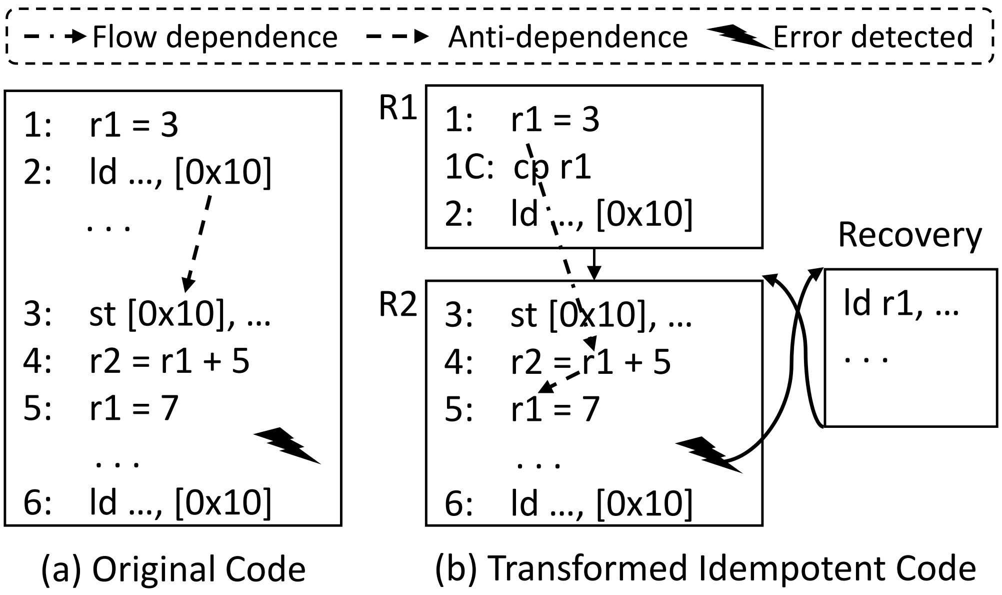


Figure 3.3: Idempotent recovery.

For a region of code to be idempotent, the inputs of the region must not be overwritten, i.e., no anti-dependence [63] on the inputs during the region execution; both memory and register inputs must be preserved to assure the side-effect-free re-execution. Figure 3.3 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 3.3(b), we thus split the code into 2 regions to break every memory anti-dependence, ensuring that memory inputs are never overwritten [29].

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 3.3(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 3.3(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 the 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[1]. That way, correct recovery is assured though $r1$ is overwritten at line 5 in the figure.

Figure 3.4 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 3.4. Similarly, an updated value at a point can be live-out to multiple region entries, e.g., $r3$ in the figure.

3.4 Idempotent Recovery Challenges for GPUs

Unfortunately, all prior works including Bolt [53] cannot be used for GPUs due to correctness/performance problems.

- Lack of store buffer exposes checkpointing store costs on the critical path.

[1]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 5.1), and (2) jumps back to the beginning of the region.

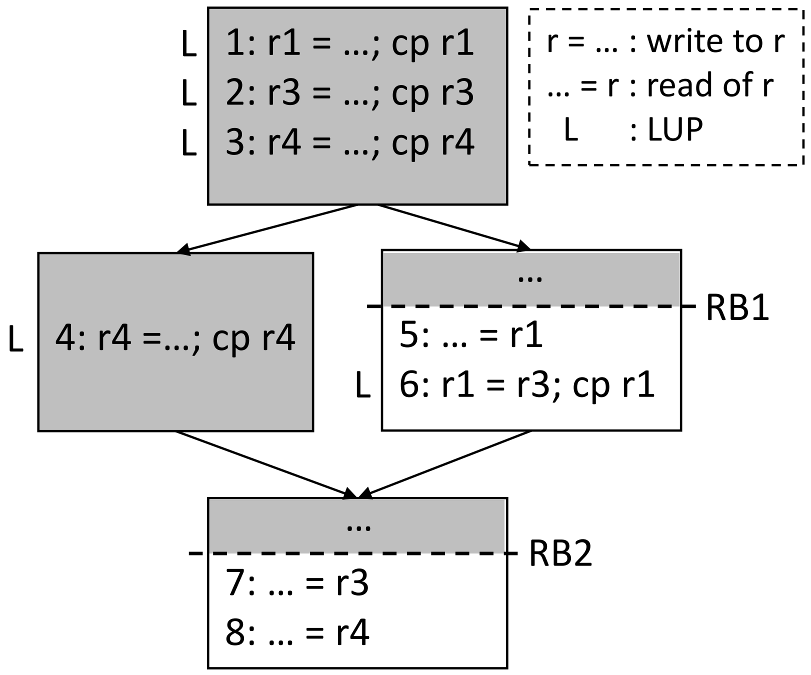


Figure 3.4: Eager checkpointing.

- Simple in-order architecture of GPU is not efficient for scheduling memory operations. Instead, they rely on context switching. But still, for memory-intensive applications, the store latency may not be hidden.

- While compiler optimizations for CPU balance for performance between register usage and stack spill, GPU resource usage management is more complicated. GPUs not only have separate share and global memory with different capacity and latency, but a shared resource such as register file and shared memory are also closely related to the occupancy (degree of parallelism) of the GPU. Using more register of shared memory may save the spill to the global memory but may also degrade the occupancy.

- While CPUs can use store buffers to hold the store being committed to cache, GPU must use a software approach to prevent register overwriting. The software solution must both achieve the correctness of recovery and low execution overhead.

3.4.1 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 3.4, 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.

3.4.2 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 6.2.1).

Given that soft errors rarely occur (0.5/day in Titan GPU, Chapter 2.3), 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 [6]: (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.

Chapter 4

Correctness of Recovery

In this chapter, we first prove how Penny correctly recovers from register file errors combining error detection codes (EDCs) with idempotent recovery. Then we provide software schemes to prevent checkpoint overwriting for correct recovery.

4.1 Proof of Safe Recovery

All prior idempotent recovery schemes require that errors must be detected within the same region where they occur; due to error propagation behaviors [49, 34], 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 [70].

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 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.

4.1.1 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 4.1.1. *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 4.1.2. *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.

4.1.2 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 4.1—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.

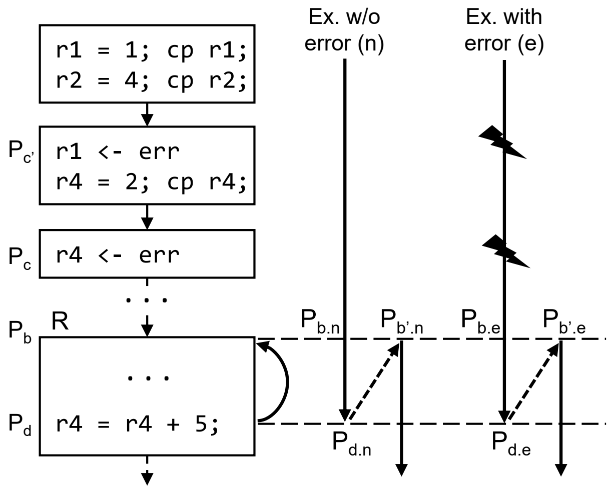


Figure 4.1: Safely recovering from errors across regions

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 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 4.1.3. *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.

$$RF(P_{b.n})|_{live(R)} = RF(P_{d.n})[CP(P_{d.n})|_{livein(R)}]|_{live(R)} \tag{4.1}$$

$$= RF(P_{d.e})[CP(P_{d.e})|_{livein(R)}]|_{live(R)} \tag{4.2}$$

$$= RF(P_{b'.e})|_{live(R)} \tag{4.3}$$

Equation 4.1 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 4.2 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 4.1.1, 4.1.2). Lastly, Equation 4.3 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 3.3). □

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

Lemma 4.1.4. *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.

$$MEM(P_{b.n})|_{live(R)} = MEM(P_{d.n})|_{live(R)} \tag{4.4}$$

$$= MEM(P_{d.e})|_{live(R)} \tag{4.5}$$

$$= MEM(P_{b'.e})|_{live(R)} \tag{4.6}$$

Equation 4.4 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 4.5 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 4.1.1, 4.1.2), all memory values remain intact, i.e., $MEM(P_{d.n}) = MEM(P_{d.e})$, regardless of errors. Finally, Equation 4.6, 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. $\square$

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

Lemma 4.1.5. *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 4.1.1, 4.1.2, 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 $CP(P_{b.n})|_{livein(R)} = CP(P_{b'.e})|_{livein(R)}$. $\square$

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.

4.1.3 Correctness in Multi-Threaded Execution

Penny guarantees safe recovery for concurrent multi-threaded executions, for race-condition-free programs.

The error propagation limit also indicates that RF errors of one thread cannot be propagated to others through shared caches before they are detected and corrected. So Penny do not have to worry about RF errors propagating across threads in multi-threaded execution.

However, the validity of the memory idempotency—used in Equation 4.4 of the safeness proof—can be violated in multi-threaded execution. This can happen in the case that after thread T_1 in region R loads from a memory address A which is a live-in of the region, another thread T_2 stores and overwrites the value at the same address before T_1 finishes the region R. If an error is detected in T_1 before the end of the region R, the region must be re-executed, but the re-execution will be incorrect since the live-in value of address A entering R has been overwritten. So these load and store access to the same address from different threads can be seen as an *inter-thread memory anti-dependence*, which may cause a violation in memory idempotency.

For a race-condition-free program, the programmer is responsible for using synchronization instructions, such as barriers, memory fences, and locks, to prevent data-races. Penny places additional region boundary at these instructions to cut inter-thread anti-dependences.

Figure 4.2 shows two example of inter-thread anti-dependence. In (1.a), a store in thread $T1$ happens after a load from the same address in $T0$, but before the region

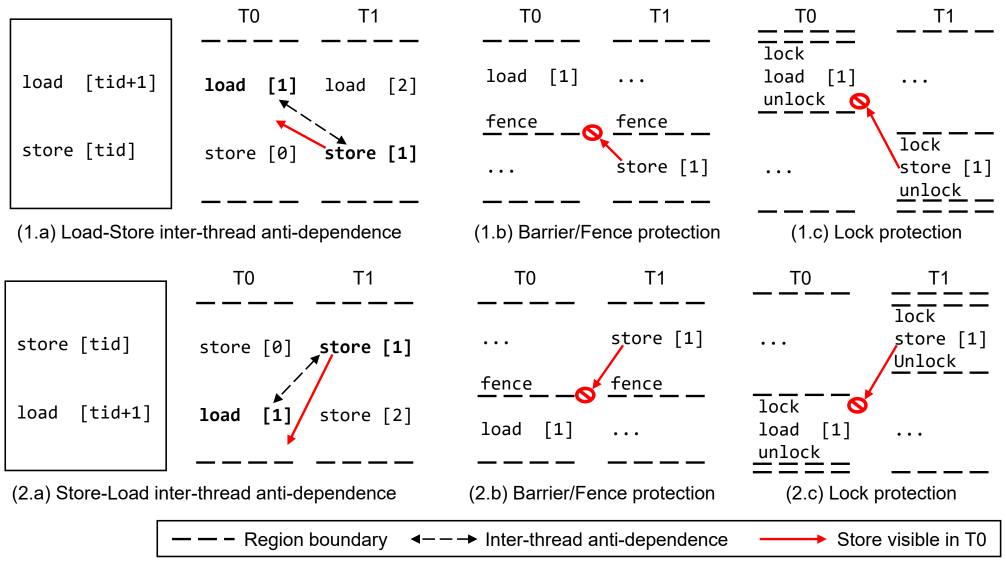


Figure 4.2: Handling inter-thread anti-dependence

ends, overwriting the value that was stored before starting the region. So, if $T0$ tries to recover from an error and re-execute the region, the idempotency is violated. If two threads are placed in different warps, this can also happen even if the store proceeds the load, such as in (2.a), or the store is in a different region.

To prevent such data-races, programmers are expected to put barriers or memory fences or ensure mutual exclusion by using locks or atomic instructions. Penny places additional region boundary at all of these synchronization instructions to protect idempotency. (1.b) and (2.b) explains how the additional boundary at the fence/barrier ensures idempotency: because the region is split at the fence, the store in $T1$ cannot pass across the boundary. In case of using a lock as in (1.c) and (2.c), the critical section becomes a separate region, and the execution of two regions including load and store becomes mutually exclusive. As a result, the store in $T2$ is prohibited from being executed during the region including load in $T0$.

4.2 Preventing Checkpoint Overwriting

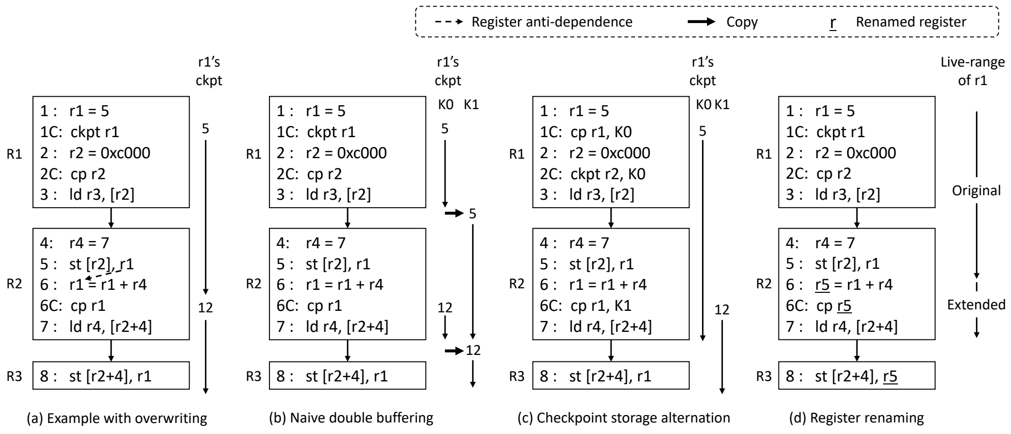


Figure 4.3: Checkpoint overwriting and prevention techniques.

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 4.3(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.

This happens because the live-in value for the current region must be preserved until the end of the region, but live-out values of the current region—that are required for the following regions—are required to be checkpointed. Bolt solved this problem by modifying the hardware store buffer to a gated store buffer (GSB). The gated store buffer the checkpointing stores and delays them being committed to cache until the end of the region. The compiler is responsible to guarantee the GSB is not overflowed at runtime by limiting the region size. However, GPUs do not comprise store buffers so we suggest several software mechanisms prevent the checkpoints from being overwritten.

The first naive approach one can think of is to imitate the gates store buffers in software using two checkpoint storages. Figure 4.3(b) shows the example of such a double buffering algorithm. Two checkpoint storages $K0$ and $K1$ are allocated. While executing a region, all checkpoints are stored to the front storage $K0$ and at the end of each region, the buffered checkpoint values stored at $K0$ are copied to the back storage $K1$. While an error happens within a region, checkpoints from $K1$ can be used for recovery. This naive double-buffering imposes a significant overhead due to a high number of unnecessary copies since the store costs are the main factor of performance overhead in Penny. For each checkpoint occurrence two stores are required: one checkpoint store to the front storage and one copy from front to back storage.

To reduce the checkpointing cost, we suggest two optimized overwrite prevention technique called checkpoint storage alternation and register renaming. The storage alternation uses two storage spaces as in naive double buffering but minimizes the number of stores by alternatively storing checkpoints to every two storages to eliminating unnecessary copies. Figure 4.3(c) shows an example of storage alternation. 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 $R2$. Penny also provides storage optimization to reduce the total checkpoint storage, considering that not every checkpointed register requires two storages.

Figure 4.3(d) shows an example of register renaming. To prevent the values being checkpointed into a same storage, the register name $r1$ at line 6 can be renamed to an unused register $r5$. The following register references that use the value (i.e. line 8) should also be renamed to $r5$.

4.2.1 Register renaming

The register renaming to prevent checkpoint overwrite similar to register renaming in the idempotent translation scheme of De Kruijf *et al.* [29]. However, targeted registers for renaming differs. In De Kruijf's algorithm, all live-in registers with anti-dependences are renamed. But Penny only needs to rename the registers that actually have their checkpointed value overwritten. For example, in Figure 4.4, both $r1$ and $r2$ values

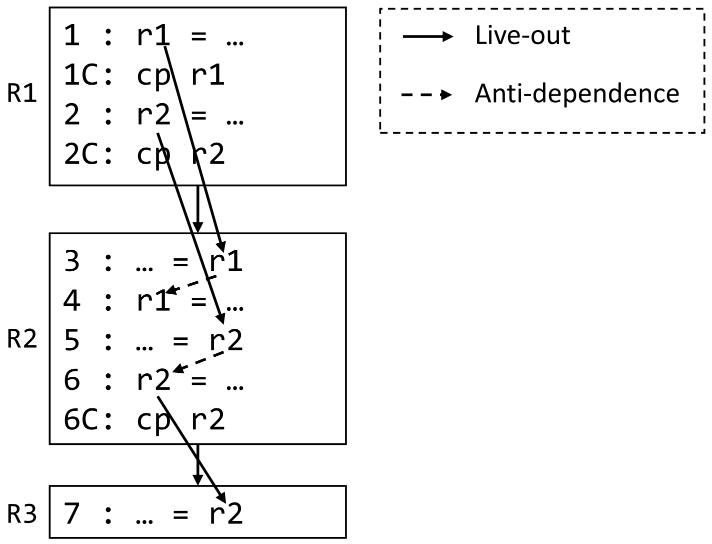


Figure 4.4: Register renaming.

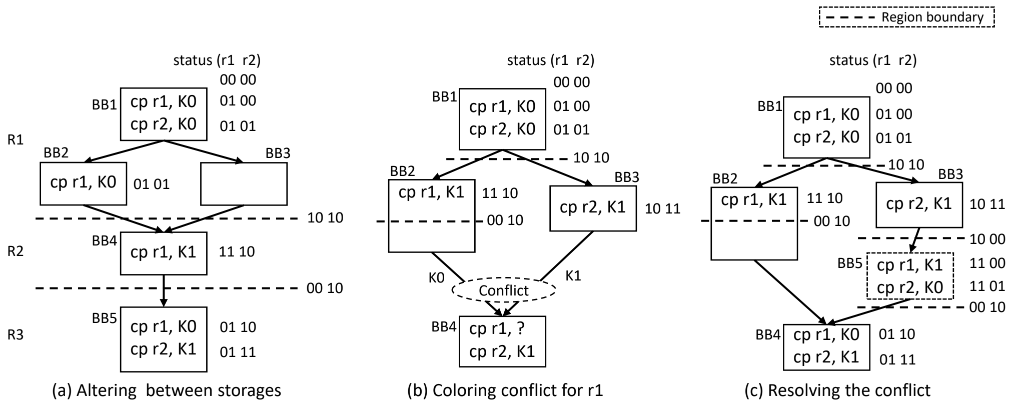
updated in $R1$ is live-in to $R2$. The checkpoints in $1C$, $2C$, and $6C$ are only used for Penny and not for De Kruijf's idempotent translation. The De Kruijf's algorithm does not rely on checkpoints, register writes which overwrites the live-in values must be renamed. For example, in $R2$, both $r1$ and $r2$ have live-in values and they are overwritten (e.g. have anti-dependence) so the registers in live 4 and 6 and their references must be all renamed. On the contrary, Penny only requires renaming for $r2$ in $R2$. This is because the value of $r1$ in line 4 is not a live-out of the regions so not checkpointed. The value of $r2$ in live 6 is used at $R3$, so it is checkpointed at line $6C$ and may overwrite the checkpointed value at line $2C$.

The register renaming is applied by modifying the live-range of the logical registers before register allocation. In the example of Figure 4.3(d), the original live range of $r1$ is artificially extended. The register allocator respects the extended range and assigns an exclusive physical register to the renamed registers. 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.

4.2.2 Storage Alternation by Checkpoint Coloring

Assigning checkpoint storage at runtime causes unrequited overhead for management, so Penny's compiler statically allocates storages to each checkpoint on compiler time. Assigning storages to each checkpoint can be seen as a simple 2-coloring algorithm. Also, applying storage alternation on all registers causes unnecessary storage and runtime overheads. Thus, our compiler first identifies the registers that have at least one checkpoint overwriting and feed them as inputs to 2-coloring. If there is no overwrite of live-in in any region, the register can be omitted from storage alternation and can be stored to a single checkpoint without altering. This helps to reduce the checkpoint storage size, which can be a limiting feature for the performance if it is stored on a limited resource such as shared memory. The compiler visits basic blocks in topological order and colors the checkpoint storages of the input registers.

If there is no overwrite of live-in in any region, the register can be omitted from storage alternation and can be stored to the same checkpoint every time. This helps to reduce the checkpoint storage size, which can be a limiting feature for the performance if it is stored on a limited resource such as shared memory.



Region boundary
status (r1 r2)
BB1 cp r1, K0 cp r2, K0
00 00 01 00 01 01
R1
BB2 cp r1, K0 01 01
BB3
10 10
R2
BB4 cp r1, K1 11 10
00 10
R3
BB5 cp r1, K0 cp r2, K1 01 10 01 11
(a) Altering between storages
BB2 cp r1, K1 11 10 00 10
BB3 cp r2, K1 10 11
K0 K1 Conflict
BB4 cp r1, ? cp r2, K1
(b) Coloring conflict for r1
BB5 cp r1, K1 cp r2, K0 10 00 11 00 11 01 00 10
BB4 cp r1, K0 cp r2, K1 01 10 01 11
(c) Resolving the conflict

Figure 4.5: Coloring checkpoints.

Basic Coloring Rules

First, we define conditions that the coloring result must satisfy to not overwrite the checkpointed values required for recovery. Figure 4.5(a) shows the example of two register $r1$ and $r2$ being colored.

- Checkpoints of a register in a region must be assigned to a same storage. Because of the control divergence, there can be multiple last update points of a register in the same region, and they must be colored in the same color. In the example, two checkpoints of $r1$ in $BB1$ and $BB2$ are labeled in the same storage $K0$, because they are in the same region $R1$. Thanks to the unified coloring, when a recovery is required in the following regions, the value can be restored from $K0$ regardless of the execution path ($BB2$ or $BB3$) taken without additional path tracking.
- For neighboring regions checkpoint must be stored into alternate storage not to overwrite the previous ones. For example, $r1$ was store in $K0$ for $R1$, so for $R2$ it is stored in $K1$ and $K0$ for $R3$.
- However, if there are no checkpoints in the region, the storage must not alternate. For $r2$ in the example, checkpoints are stored to $K0$ for $R1$, and the next region must checkpoint to $K1$. However, since there is no checkpoint of $r2$ in $R2$, the next region $R3$ must not alternate back to $K0$, but still checkpoint to $K1$. Otherwise, e.g. if the $R3$ checkpoints $r2$ to $K0$, it may overwrite the live-in checkpointed value from $R1$ which is required for recovery when an error happens before the end of $R3$.

To satisfy these conditions, when the compiler colors the checkpoints in the code sequentially, it must remember if there a register has been checkpointed in a region as well as the last labeled storage. Thus, Penny uses a two-bit representation to remember to current coloring status for each colored register: *storage bit (S)* and *flip bit (F)*. The storage bit S_r tells which storage the upcoming checkpoint of r must be stored and the flip bit F_r presents if the register r has been checkpointed in this region. When a checkpoint or a region boundary is met during coloring following rules are used to color the checkpoint and update the coloring status:

- **On a checkpoint of r:** Label the checkpoint with S_r and set 1 on the flip bit F_r to notify the next region must alternate the storage. *cp* $r, S_r; F_r \leftarrow 1$.
- **On a region boundary:** If the flip bit F_r is set, alternate the storage for next region by flipping S_r and reset F_r to 0. The xor operation ($\oplus$) can be used to flip the storage bit. $S_r \leftarrow S_r \oplus F_r; F_r \leftarrow 0$.

In Figure 4.5, we mark the coloring status of each register in the order of SF on the right of the basic blocks. For the example in (a), both registers start with an initial status of 00. When checkpoint of $r1$ and $r2$ are met in $BB1$, the checkpoint is labeled with current storage value 0 in S and F bit for each register is set. In $BB2$, the checkpoint $r1$ is labeled with the same storage as with the checkpoint in $BB1$. On the region boundary after $BB2$ and $BB3$, since both F bit of two registers are 1, the S bit value is alternated from 0 to 1. In $BB4$, $r1$'s checkpoint is assigned to storage $K1$ and F bit is set. Since only $r1$ is checkpointed in $R2$, e.g. only the F bit of $r1$ is set, only the S bit of $r1$ is altered at the end of $R2$. Thus, in $R3$, $r1$ is checkpointed to 0 and $r2$ is checkpointed to 1.

Resolving a Coloring Conflict

Due to a control-flow divergence, the coloring status may differ over multiple incoming paths at a convergence point. Figure 4.5(b) 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. Note that not only the storage status S but also the flip status F must be unified to the same status to guarantee the following checkpoints to not overwrite required values. For the left path, the coloring status for two registers was 0010 and 1011 for the right, and these values must be unified to the same status value.

To ensure the same color at the convergence point no matter which path is taken, Penny inserts a new adjustment block, that has a dummy checkpoint and region boundaries for a conflicting register, over one or more paths to the point. The dummy checkpoints simply checkpoint the current live-out value of the region to the opposite storage of the latest storage used to match status between paths. As shown in Figure 4.5(c), due to a new block $BB5$, the colors in the 2 paths to $BB4$ are both $K1$. Depending on the coloring status, new region boundaries must also be inserted. These inserted region boundaries affect all colored registers, so Penny must consider the status of all live registers at once when unifying the status and inserting adjustment block. For example, in the original application code in Figure 4.5(b), only the status of $r1$ had a conflict, but when the adjustment block is inserted as (c), the $r2$ also requires a dummy checkpoint. After the adjustment block insertion, the coloring status for both paths becomes 0010.

In Figure 4.6, execution path from three basic block $BB1$, $BB2$, and $BB3$ are converged into $BB4$. Output coloring status of each converging path for register $r1$ to $r4$ are 01100011, 01001101, and 01100000 respectively. Let's assume we want to unify the merged coloring status into 01100000. In such a case, one region boundary is needed to be inserted to the convergence path from $BB1$ and an adjustment block with checkpoints to $r2$ and $r4$ interposed by two new region boundaries. By applying the coloring status transition rules, it could be seen that the output status from $BB1$ and $BB2$ are translated into unified status after passing the inserted region boundaries and checkpoints.

In order to reduce the overhead of inserted checkpoints for adjustment block, Penny tries to choose the unifying status that produces the minimum number of checkpoints. For all possible merged coloring status, the total number of added checkpoints for all

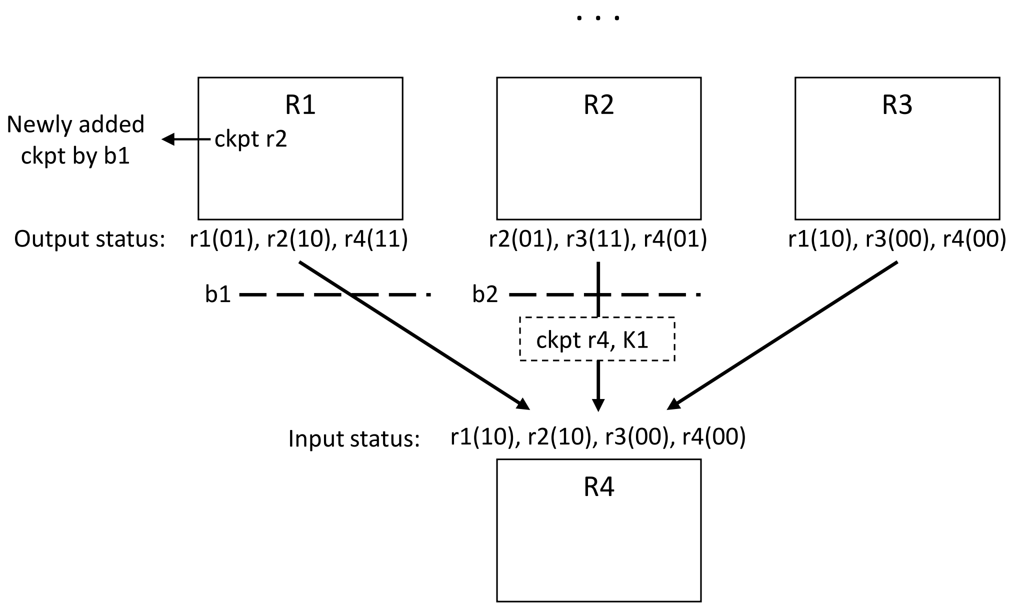


Figure 4.6: Merging the coloring status.

paths are calculated, and the coloring status with minimal cost is selected.

In some rare cases, due to the added region boundary, additional live-out value has to be checkpointed. For example in the given code, because of the newly added $RB1$, another value of $r2$ has to be checkpointed in $BB1$. We call this a *collateral checkpoint* added by the adjustment code. The newly added collateral checkpoint may change the original coloring status of its path, i.e. the output status of $BB1$, and current coloring status may be invalidated, i.e. current coloring is no more a safe solution.

In such a case, Penny re-colors the graph from the beginning with the added adjustment codes and collateral checkpoints. However, naively applying re-coloring whenever the coloring safeness is violated may result in unterminated iteration: the coloring status may oscillate between a number of statuses endlessly. To prevent this, the merging algorithm of coloring status must guarantee convergence. So when the graph is re-colored, the unification status at a conflict point is selected in a non-regressional way: previously added adjustment codes are not removed and new boundaries are added for status merging.

Note that added adjustment checkpoints have a high chance of getting safely removed in the pruning phase (Section 5.5), and therefore the resulting overhead is not significant in the majority of applications we tested (Section 6.2.3).

The Comprehensive Coloring Algorithm

Each labeled register has its labeling status bits and the labeling algorithm labels all the registers in the same pass. The labeling algorithm traverses over the CFG blocks in a reverse post order and labels checkpoints in each block at a time. Labeling a block is done in three steps; first, merge labeling status from the output of the parent block into an input status. Only non-back-edges are merged. Second, the body of the block is labeled following the basic labeling rules. For the last, if there are outgoing back-edges from the block, their statuses are merged back into child blocks. After the first and third steps, if Penny decides there are inconsistencies brought by new adjustment region boundary, labeling is re-done from the beginning.

4.2.3 Automatic Algorithm Selection

It is not a trivial task to select between the two overwrite prevention techniques. The register renaming may increase the register pressure and diminish the occupancy, while the storage alternation may increase the checkpointing cost.

Thus we provide an automatic optimization selection module to choose the better between the two. Penny compiler the code using both techniques and estimate the final cost of the generated code in a similar way to the one in Section 5.2 to pick the best. For the generated code, load and store instructions are given higher cost than arithmetic instructions and they are weighted depending on the depth of the loop they are placed.

4.2.4 Future Works

Currently, either of register renaming and storage alternation is selected for compiling each kernel. In the future, we can suggest techniques that selectively apply both techniques in a more fine-grained manner: some checkpoints are renamed and some checkpoints use storage alternation for optimal performance.

Another idea is that instead of the two-coloring algorithm, we can use a more generalized coloring algorithm similar to that used in the register allocation in compilers [18]. The main challenge is how to properly set the interfering edges of the graph since the interference semantics for overwriting case is different from the register allocation and decide when to coalesce and spill the graph.

Chapter 5

Performance Optimizations

5.1 Compilation Phases of Penny

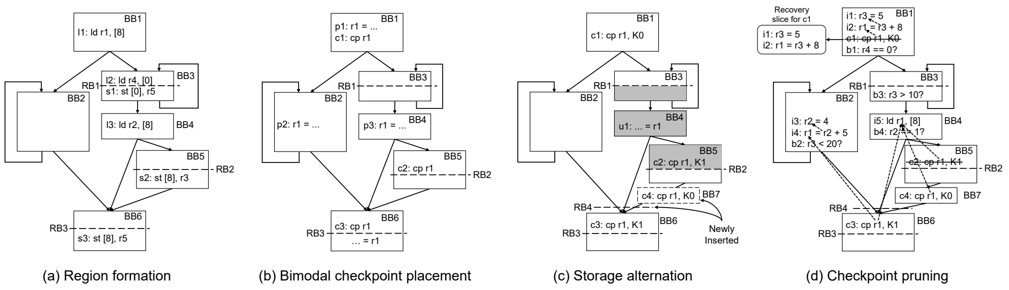


Figure 5.1: Compilation phases of Penny.

This section provides a high-level overview of Penny's compilation workflow for generating the final checkpoint-enabled code. 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. The goal of the compiler is to produce correct yet low-overhead code.

Figure 5.1 demonstrates the main phases in the compilation.

5.1.1 Region Formation

Penny first partitions the entire program into idempotent regions by breaking every memory anti-dependence to prevent their memory inputs from being overwritten.

Figure 5.1(a) shows 4 memory anti-dependences in the code, i.e., $(l1, s3)$, $(l2, s2)$, $(l3, s2)$, and $(l3, s3)$ where l and s form the load-store pair. There can be multiple region formation solutions for cutting all anti-dependence paths in the code. Figure 5.1(a) shows one solution with 3 region boundaries of $RB1$, $RB2$ and $RB3$.

To minimize the number of region cuts, De Kruijf et al. [29] uses a heuristic algorithm to minimize the number of cuts. This algorithm is both heuristic and not aware of checkpoint costs, i.e. does not minimize the number or cost of the checkpoints being inserted. So we provide an optimal checkpoint-aware algorithm to minimize the checkpointing cost.

Once region boundary is determined, last update points (LUPs) of each region's live-out [63] registers are discovered as the candidate of values to be checkpointed.

5.1.2 Bimodal Checkpoint Placement

While Bolt [53] 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 5.4). With that in mind, we perform a checkpoint scheduling to reduce the estimated cost of inserted checkpoints.

For example, in figure 3.4, there are two checkpoints for $r1$ in $R1$, and if the left path is taken at the branch, unnecessarily two checkpoint instructions for $r1$ transpire. However, we relax it by exploiting the fact that each checkpoint can be delayed until the end of the region. 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.

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 (boundary). The later *local checkpoint scheduling* step tunes the bimodal schedule for better performance in the code generation phase.

For each region boundary in Figure 5.1(b), all LUPs for $r1$ can be found by backward-traversing all execution paths until the update on $r1$ is found. We focus on region boundaries instead of the range of regions, because the latter can dynamically change depending on the execution path. Point $p1$ is the LUP for boundary $RB1$, and $p3$ is for $RB2$. $RB3$ has two LUP $p2$ and $p3$. The bimodal placement places checkpoint at either end-point of these mappings. For example, checkpoints are placed either at $RB3$ or both at $p2$ and $p3$. Checkpoints $c1$, $c2$ and $c3$ in Figure 5.1(b) is the result of bimodal placement. For the region boundary $RB1$, checkpoint is placed at $p1$ instead of $RB1$, because it is expensive to place the checkpoint inside the loop whose body is a basic block $BB3$. In contrast, for regions ending at $RB3$, checkpoint is placed at $RB3$ to avoid checkpoint insertion at the LUP $p2$ inside the loop ($BB2$). Note that $c3$ servers for checkpointing $r1$'s values from both $p2$ and $p3$.

5.1.3 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 4.2 provides the details.

Due to the lack of store buffers in GPU, a checkpoint value can be overwritten before it is used for recovery. To ensure that no necessary checkpoint is overwritten, we introduce two techniques in Section 4.2: register renaming and 2-coloring for storage alternation. The storage alternation must be applied before checkpoint pruning.

Figure 5.1(c) shows such an example in the shaded region that starts from $RB1$ and ends at $RB2$. For an execution passing through this region, $r1$'s value is checkpointed at $c1$ and subsequently overwritten by another checkpoint at $c2$. If an error then occurs before $RB2$, i.e., the end of the region, its re-execution starting from $RB1$ will lead to incorrect recovery. That is because the input of the region, i.e., live-in register $r1$ checkpointed at $c1$, has already been overwritten.

With that in mind, we use two storages $K0$ and $K1$ alternatively to checkpoint $r1$ in each region to avoid overwriting the previously-checkpointed value that will be used for the recovery of the region. For some region, if its input register, that has been checkpointed in $K0$, is updated and live-out in the region, Penny avoids using the storage $K0$, where the input exists, and checkpoints the register in the other storage $K1$. Interestingly, this checkpoint storage alternation can be reduced to a 2-coloring problem. Coloring the storage for each checkpoint can be easily achieved by performing a pre-order depth-first search during which either $K0$ or $K1$ is determined as a color.

However, the coloring might fail at a control flow convergence point if the colors of a checkpoint on incoming paths differ from each other. In Figure 5.1(c), 3 execution paths converge to $BB6$, i.e., $BB2 \rightarrow BB6$, $BB4 \rightarrow BB6$, and $BB5 \rightarrow BB6$. For the first two paths, $r1$'s last checkpoint is $c1$ that uses $K0$ as a storage. So the next checkpoint in $BB6$ must store $r1$ to $K1$ to avoid overwriting the previous value in $K0$. However, on the 3rd path ($BB5 \rightarrow BB6$), $r1$ was checkpointed at $c2$ using $K1$ as a storage. This implies that the next checkpoint in $BB6$ should be stored at $K0$. This conflicts with $K1$, the storage that must be used for the checkpoint when $BB6$ is reached through other two paths $BB2 \rightarrow BB6$ and $BB4 \rightarrow BB6$.

To resolve the conflict, Penny creates a new region ($BB7$) between the conflicting regions and inserts a compensation checkpoint ($c4$) as shown in Figure 5.1(c). In $c4$, Penny simply copies the checkpointed value of $r1$, which is available in $K1$, into $K0$. Consequently, the next checkpoint in $BB6$ uses the same storage $K1$ no matter which path is taken to reach $BB6$.

5.1.4 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.

Whether or not a checkpoint is prunable is determined by tracing back both data- and control-dependence values from the checkpoint and by verifying if the values can be recomputed at recovery time. If all dependent values can be reconstructed, Penny regards the checkpoint as prunable. In Figure 5.1(d) the data dependences tracked from checkpoints are shown in dotted arrow-lines. Consider $c3$, which has multiple paths to track the dependences. If all the data dependence can be safely restored by using constant (e.g., $i3$) or memory value that is not overwritten before the use of the loaded register (e.g., $i5$), then the checkpoint can be safely pruned. In addition to the data dependences, control dependences should be tracked and verified to be reconstructible at recovery time. For example, in case of soft error detected, to decide which path has been taken, branch condition values ($b1$, $b2$, $b3$, and $b4$) and their dependent values have to be reconstructible as well to correctly execute the recovery slice along the path. An example recovery slice that can recompute the value of $r1$ for the pruned checkpoint $c1$ is shown in the figure.

5.1.5 Storage Assignment

By default, Penny uses two checkpoint storage spaces, that are protected by ECC, to save checkpoints: shared and global memories of GPU. Its available non-ECC registers can be optionally used as checkpoint storage to improve performance at the cost of compromised reliability. Care must be taken for the storage assignment. Since low-latency GPU caches have limited size and thus assigning too many checkpoints there can reduce the occupancy. In light of this, Penny carefully distributes checkpoints to the storage spaces thereby reducing the run-time overhead. Also, Penny leverages an appropriate storage layout with coalesced memory accesses in mind.

5.1.6 Code Generation and Low-level Optimizations

As a final step, during the code generation, Penny performs several compiler optimizations to minimize the added instruction cost due to checkpoint stores and their address calculation. The optimizations include local instruction scheduling, redundant code elimination, and loop invariant code motion, etc.

5.2 Cost Estimation Model

Several numbers of Penny's optimization rely on cost models to estimate and compare the quality of the code resulting from an optimization decision. We define the *checkpoint-cost* by weighting each checkpoint based on the nested loop-depth the checkpoint is placed. Specifically, we use $2^{K \cdot d}$ as the cost of a checkpoint at loop-depth d where K is a large enough constant value to prioritize checkpoint in inner loops. We also define a *instruction-cost* for instructions added after the code generation phase, This metric can more accurately estimate the overhead because each checkpoint can be translated into an arbitrary type and a number of instructions. We define the cost of each instruction adding an additional type based weight W_t to the loop-depth based weight: $W_t \cdot 2^{K \cdot d}$. Load and store instructions have a larger constant weight compared to arithmetic instructions.

The most commonly used metric is a *checkpoint-level weight-accumulated cost* that accumulates the cost of all checkpoints, where each checkpoint cost is weighted based on its nested loop-depth. This is because checkpointing instruction added inside a deeply nested loop is the most significant source of overhead. This is formulated as $\sum_{c \in C} W_d(c)$, where C is all checkpoints in the code and $W_d()$ is a weight function that gives higher cost for instructions in deeper loop-depths. We also introduce a *instruction-*

level weight-accumulated cost to estimate the cost of all additional instruction I added after code generation. This can be more accurate than the checkpoint-level estimation because each checkpoint is translated into a series of instructions that can be in variable lengths depending on the context. We add an additional instruction-type-based weight $W_t()$ that gives higher cost for load and store instructions: $\sum_{i \in I} W_t(i) \cdot W_d(i)$.

5.3 Region Formation

To translate a code into idempotent regions, the most common way to handle memory anti-dependences is to separate the load and store of the anti-dependences into different regions. There are multiple memory anti-dependences in the code and each anti-dependence has a multiple execution paths and they are all required to be cut by region boundary. All execution paths from the load to the store must include at least one region boundary. The region boundary can be placed in anywhere on the paths and region boundary placed in a common interleaving path can be shared among multiple paths of anti-dependences. This makes the search space extremely complicated, so it is not easy to use an optimal search.

We first introduce the heuristic algorithm from De Kruijf et al. [29] and its limitations. Then we introduce our optimal region partitioning algorithm that is aware of the checkpoint costs. Additionally, we introduce a region stitching that saves the loaded memory load value instead of cutting the region to handle the memory anti-dependence, and how the stitching is combined into the optimal pruning.

5.3.1 De Kruijf's Heuristic Region Formation

De Kruijf et al. [29] uses an approximate algorithm to minimize the number of anti-dependence cut to form maximal regions. The region cut problem is translated into a vertex multicut problem which is NP-complete [37]. The problem is again reduced to a hitting set problem and an approximated algorithm [24] is used to solve it. iGPU [60] uses a slightly modified version of this algorithm for GPU, which puts the region boundary at the location with a “relatively” little live-in states.

However, this algorithm is not only an approximated algorithm but also no checkpoint-aware, i.e., it does not try to minimize the number of checkpoints being inserted.

5.3.2 Region splitting and Region Stitching

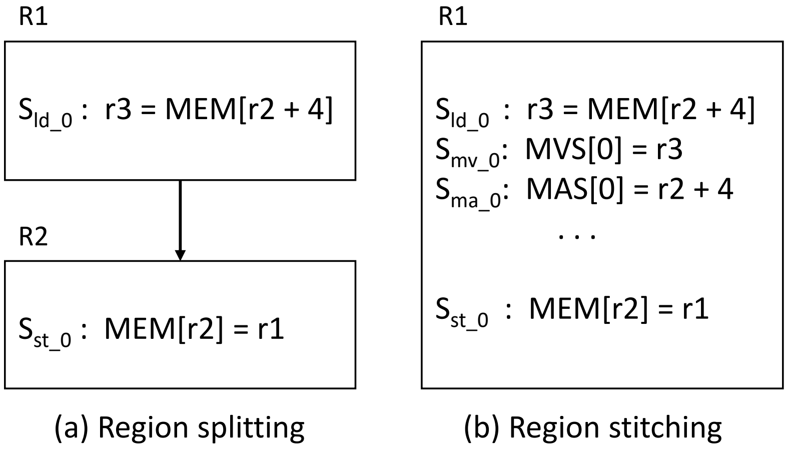

R1
S_{ld_0} : r3 = MEM[r2 + 4]
R2
S_{st_0} : MEM[r2] = r1
(a) Region splitting
R1
S_{ld_0} : r3 = MEM[r2 + 4]
S_{mv_0}: MVS[0] = r3
S_{ma_0}: MAS[0] = r2 + 4
. . .
S_{st_0} : MEM[r2] = r1
(b) Region stitching

Figure 5.2: Example of region splitting and region stitching.

In prior idempotent recovery schemes, all memory anti-dependences have to be cut, resulting in no anti-dependences to remain in the same region (Figure 5.2(a)). Penny proposes another technique to safely handle memory anti-dependences for recovery; a *region stitching*. Region stitching prevents region splitting by saving loaded memory values in a separate storage for recovery. For certain anti-dependences, split region boundary may produce too many checkpoints, so stitching up the region into a bigger one may lead to better performance.

Figure 5.2(b) is an example of how the anti-dependence memory value is preserved. Memory values are stored eagerly after the load instruction [1]. The loaded values from

[1]We chose eager saving instead of lazily saving it right before the store. The lazy scheme may result in avoiding unnecessary memory-value saving instructions, but reconstructing load addresses at the time of

memory are saved to *memory value storage* (MVS). The compiler assigns storage indies for each load with anti-dependences. The current implementation only stitches load instructions out of the loop, because the saved instruction might be overwritten in loops.

Unlike previous work [35], Penny does not store the memory address of the load instruction. Instead, it is computed at recovery time using either of the two techniques we propose. The compiler can generate a recovery slice that only consist of instruction dependent to compute the memory address of the required load instruction. At recovery time, this recovery slice is executed to compute the memory address value, and from the address and memory value of MVS, the memory value is restored. Then the program can be safely executed starting from the beginning of the region. The second method is an exception handling can be used if there is hardware support for this. At recovery time an exception is placed right before the load instruction S_{ld}, and the execution is re-started from the beginning of the region. When the execution hits the exception, the exception handler can get the value of the calculated memory address (i.e. $r2 + 4$). The memory value is restored from MVS and the execution can be safely resumed.

5.3.3 Checkpoint-Cost Aware Optimal Region Formation

There can be an excessive number of region formation decisions considering which anti-dependence to handle with region stitching and where to put region boundaries. Multiple regions can share common checkpoints as their live-ins, so checkpoint cost cannot simply be computed for each region-cut and accumulated.

Penny provides an efficient algorithm that finds a minimum cost region formation combining both region splitting and region stitching.

---

the store is usually difficult or impossible. Note that due to limitation of alias analysis, actual load and store address of an anti-dependence might be different.

Algorithm Overview

We found that using a greedy algorithm or finding each boundary using a local minimum does not lead to a sufficient global solution. Unlike register renaming, which is hard to estimate the exact spilling cost, the number of checkpoints can be precisely computed from the boundary decision, which directly translates into the overhead cost. So aggressive optimization to minimize the checkpoints is expected to give an evident improvement in performance.

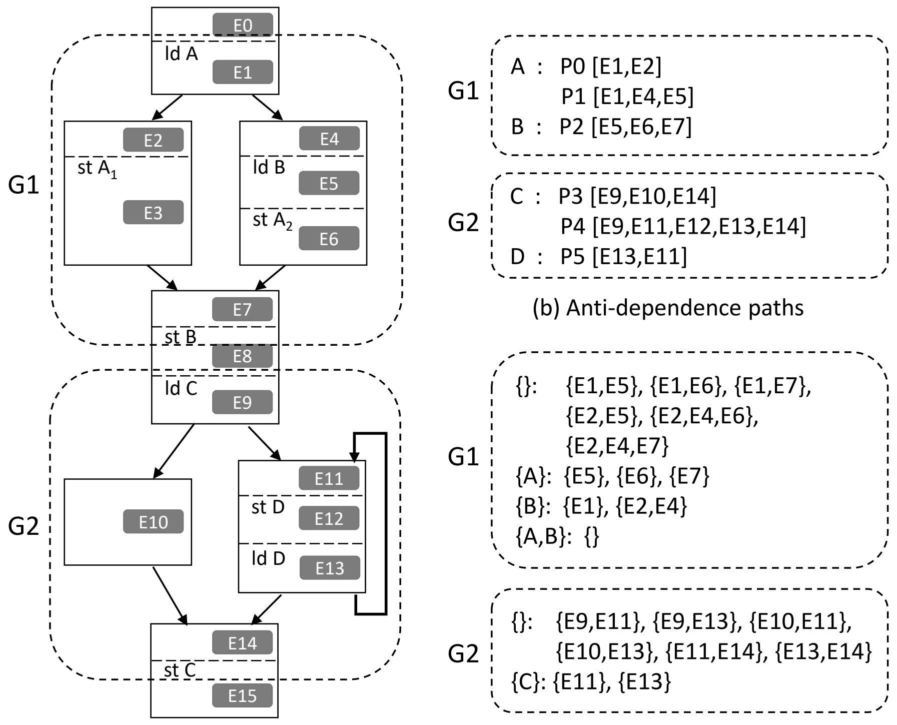
Search space for comparing all possible combinations of region stitching and boundary placement can be exceedingly large. Penny provides a technique to decompose the region formation problem into multiple levels to reduce this computation. For each divided sub-section, only the best T solutions are picked and merged into the upper level in order to limit the number of merging computations.

Problem Decomposition

First the partitioner decomposes the problem, as in example presented in Figure 5.3. Figure (a) shows a CFG with all anti-dependences (ldA, stA_1), (ldA, stA_2), (ldB, stB), (ldC, stC) and (ldD, stD). For convenience, basic blocks are split by load and store instructions of anti-dependences and divided into *segments* (e.g. $E0$ to $E15$). Then all possible paths for each anti-dependence are discovered. In figure (b), all possible paths for each anti-dependence in enumerated from $P0$ to $P5$ labeled using a sequence of segments.

Now all the paths are clustered into groups. If a path intersects with another, i.e. has a common segment, they are put into the same group. In this example, paths are grouped into two groups $G1$ and $G2$. Each group does not have an intersecting path, so region splitting can be solved individually for each group.

For each group, all possible region stitching is applied, and after removing the stitched anti-dependence all possible region splitting solutions are computed in segment-level. In other words, the decision is made in an abstract form that on which segments



(a) Anti-dependences on a control flow graph

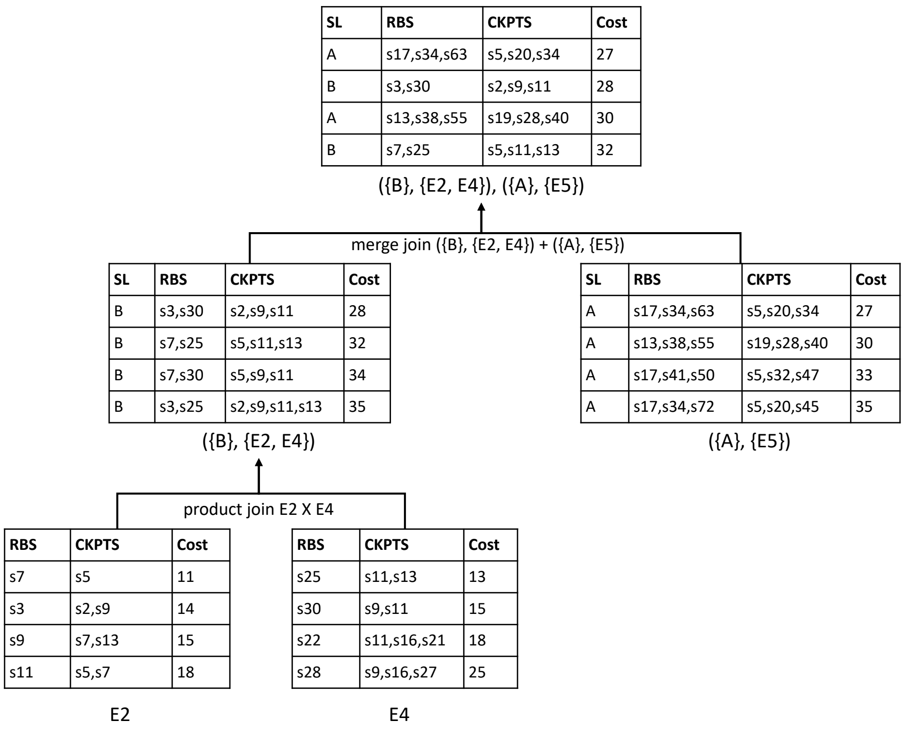
(b) Anti-dependence paths

(c) Segment-level region-formation solutions

Figure 5.3: Example of decomposing the region-cut problem.

the regions-cuts are placed, and the actual instruction-level position is not considered in this step.

Figure (c) shows the region formation result for $G1$ and $G2$. The left part of the list shows a set of loads instructions that stitching is applied. The right part is the list of all possible region-cut solutions. Let's articulate solution with T stitching set and region-cuts C as $(T : C)$ For example, $(B : E2, E4)$ in figure (c) presents that after anti-dependence B is stitched, remaining paths $P0$ and $P1$ can be cut by placing region-cut at $E2$ and $E4$. All possible cutting set solutions are computed and only the minimum set is kept. To get a minimum solution set, all solutions $A = (S_A : C_A)$ can be eliminated iff, there exists another solution $B = (S_B : C_B)$ such that $S_B \subseteq S_A$, $C_B \subseteq C_A$ and $A \neq B$


({B}, {E2, E4}), ({A}, {E5})
merge join ({B}, {E2, E4}) + ({A}, {E5})
({B}, {E2, E4})
({A}, {E5})
product join E2 X E4
E2
E4

SL	RBS	CKPTS	Cost
A	s17,s34,s63	s5,s20,s34	27
B	s3,s30	s2,s9,s11	28
A	s13,s38,s55	s19,s28,s40	30
B	s7,s25	s5,s11,s13	32

SL	RBS	CKPTS	Cost
B	s3,s30	s2,s9,s11	28
B	s7,s25	s5,s11,s13	32
B	s7,s30	s5,s9,s11	34
B	s3,s25	s2,s9,s11,s13	35

SL	RBS	CKPTS	Cost
A	s17,s34,s63	s5,s20,s34	27
A	s13,s38,s55	s19,s28,s40	30
A	s17,s41,s50	s5,s32,s47	33
A	s17,s34,s72	s5,s20,s45	35

RBS	CKPTS	Cost
s7	s5	11
s3	s2,s9	14
s9	s7,s13	15
s11	s5,s7	18

RBS	CKPTS	Cost
s25	s11,s13	13
s30	s9,s11	15
s22	s11,s16,s21	18
s28	s9,s16,s27	25

Figure 5.4: Joining solutions.

Joining Partial Solutions into a Global Solution

The globally minimum solution can be computed by finding the best region cut positions for each segment and merging them into a final solution, in the reverse order of decomposition. Figure 5.4 shows an example of merging.

In the first step, for each segment in the segment-level solution, all region boundary placements are tried to compute the best checkpointing resultCheckpointing cost is computed by checkpoints produced by each boundary placement. Checkpoints are weighted by loop depth it belongs and summed into a score. With these checkpoints inside loops are more likely avoided than the ones outside loops. Penny uses a sorted table to keep the best T results. Each item in the table consists of fields including stitched loads (SL), region boundaries (RBS), checkpoints (CKPTS), and the cost value.

Each component results are merged into a global solution in multiple steps. For each step, Penny keeps the best T results in a sorted table. Two kinds of joining operations are used for joining partial solutions into a larger composition. A product join operation combines two tables by generating items from all possible pairs of items from each table. Stitched loads, region boundaries, and checkpoints are merged with union operation for each pair, and the cost is recomputed with the combined checkpoint set. A merge join is simply merging two lists by keeping only T best results from both lists.

The first merging step is to merge each segment into a segment-level solution by product join. Left bottom part of figure 5.4 shows merging result from $E2$ and $E4$ by a product join. After merging, region stitching cost for B are added to compute final costs for solution $(B, E2, E4)$. The cost of stitching is computed similarly to the checkpointing cost. For the second step, all possible solution inside a group is merged joined to concentrate into the best T result for the group. In this example, best results from $(, E1, E5)$ to $(A, B,)$ are sorted and selected. For the last step, results from each group are joined with the product join.

Instead of keeping just one best item for each step, Penny keep multiple best items. This is because there are set operations is involved in checkpoint merging, so a combination of the minimum results from each sub-results may not lead to a combined minimum. However, keeping only a few best T results at each step is enough to generate optimal or near-optimal solutions in most cases, so the merging operation time is ignorable.

5.4 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.

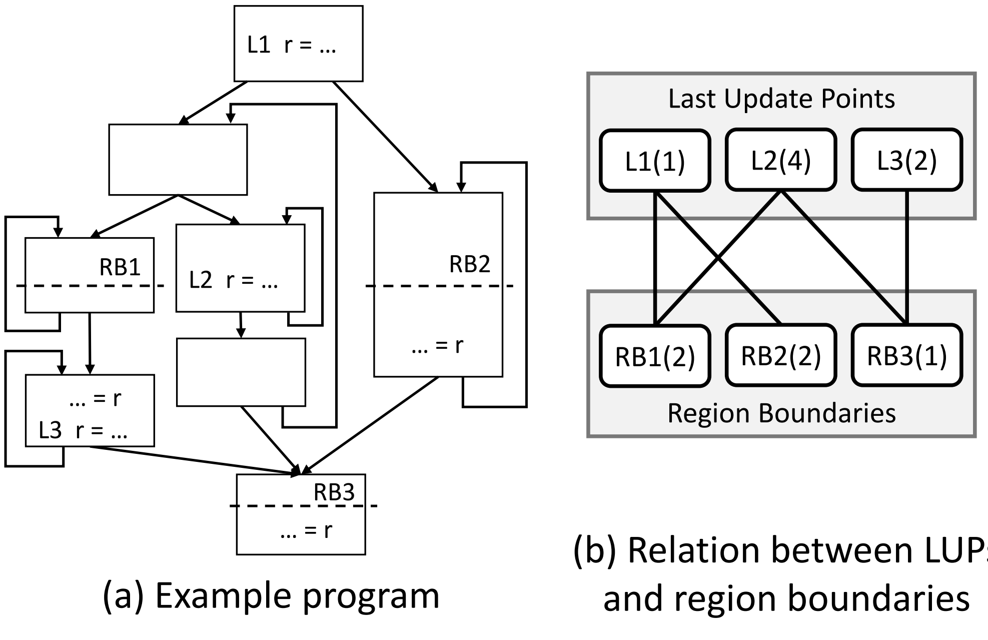


Figure 5.5: Bimodal checkpoint placement.

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 5.5(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 5.5(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 5.5(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 [24] 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 [31, 44, 23] 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.

5.5 Optimal Checkpoint Pruning

5.5.1 Bolt's Naive Pruning Algorithm and Overview of Penny's Optimal Pruning Algorithm

Bolt [53] 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.

5.5.2 Phase 1: Collecting Global-Decision Independent Status

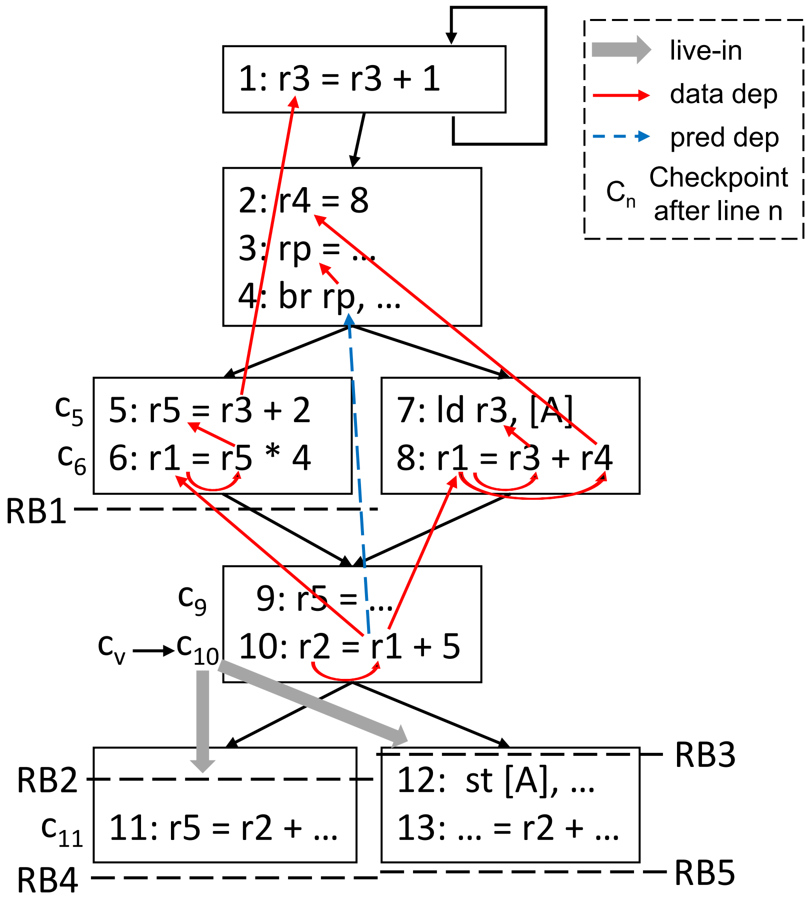


Figure 5.6: Example of a checkpoint validation.

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 [63], 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 5.6, 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 [63]. We represent predicate and data dependences in a graph and call it PDDG (predicate/data dependence graph).

As shown in Figure 5.7, 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[2] 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*.

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

[2]GPU memory is protected by ECC, and Penny ensures that register file errors never propagate to memory (See Appendix 4.1).

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 5.7. 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 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 9 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 11-12), 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 5.6, 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.

DFS backtracking and state merging: Once terminal nodes are encountered (lines 7-12 in the algorithm), DFS triggers the backtracking to propagate the validation state of non-terminal nodes being visited to their descendant (lines 13-17). For a non-terminal

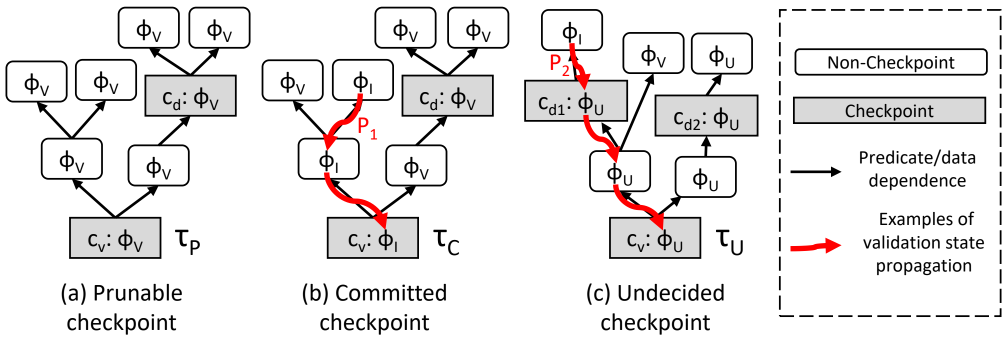


Figure 5.7: Merging validation states in PDDG.

node, Penny collects all nodes it depends on (line 14) and visits them (lines 15-17). The validation state of the dependent node is determined by merging the state it depends on (line 17), 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 5.7(a). In contrast, propagation path P_1 in Figure 5.7(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 18 of the algorithm checks if its input state is ϕ_I; if so, the state is lowered to ϕ_U (line 19). Figure 5.7(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 5.6. 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.

5.5.3 Phase2: Ordering and Finalizing Renaming Decisions

Collecting Decision Dependence between 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 5.6, 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 details the dependence analysis. COLLECT-DECISIONDEPS 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 8-9). 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 12 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 5.8 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 [76] to sort the DDG in a topological order along with identifying strongly connected components (SCCs) in a traversal. As shown in Figure 5.8, 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

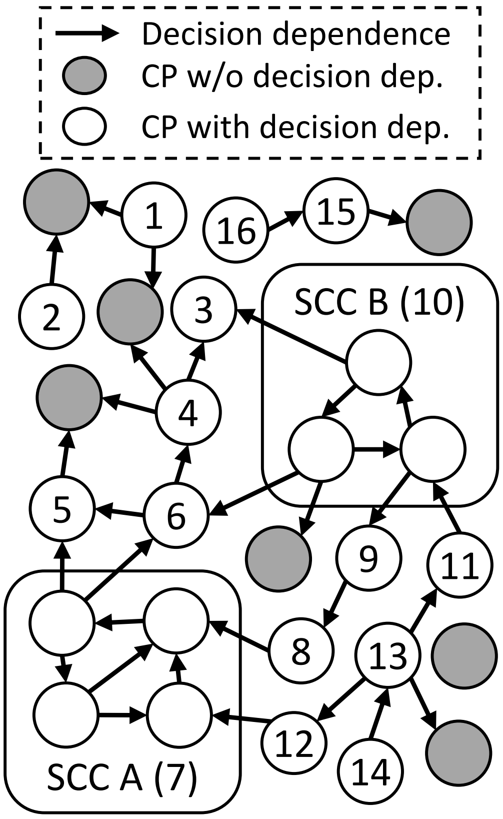


Figure 5.8: Decision dependence graph.

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 5.2); 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.

5.5.4 Effectiveness of Eliminating the Checkpoints

In this section, we analyze how effectively the pruning algorithm reduces the checkpoint by analyzing the compilation statistics. Performance evaluation on how the eliminated checkpoints reduce the overhead can be found in Section 6.2.4.

Statistics on Verification Path Tracking

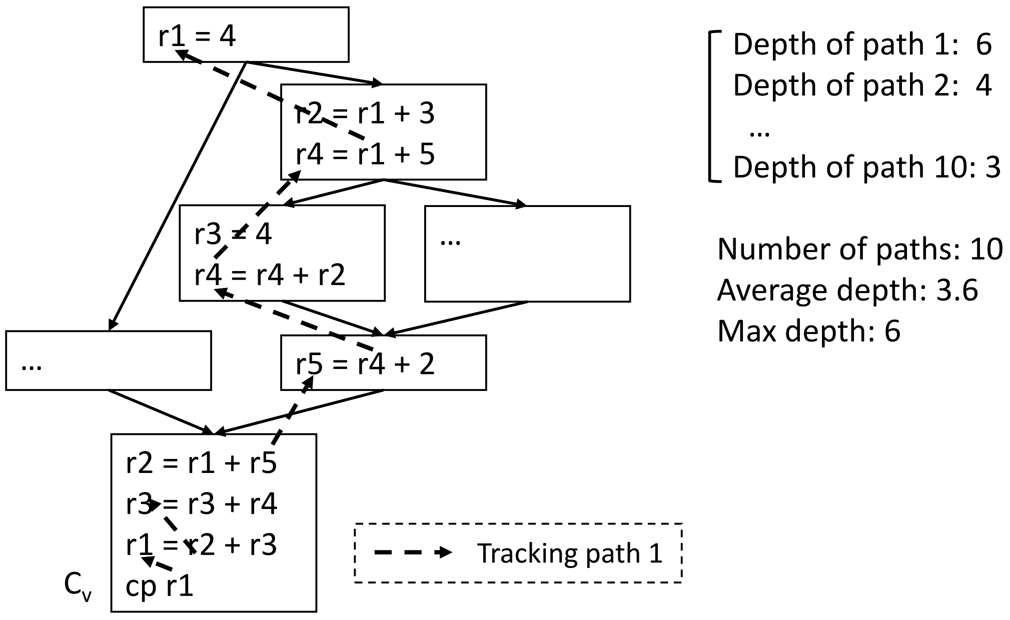


Figure 5.9: Maximum depth and average depth of instruction visited while pruning.

In this section, we analyze how deep and broadly the verification process recursively tracks down the data- and predicate- dependence in our optimized pruning algorithm. Figure 5.9 shows how the collected statistic values are defined. The verification starts from C_v and data- and predicate- dependences are tracked down recursively. The dependence tracking diverges into multiple paths because 1) there can be multiple operands in an instruction to track (i.e. for $r1 = r2 + r3$ two operands $r2$ and $r3$ must be tracked), and 2) control flow divergence may exists on the tracking path. In the given example the dependence tracking beginning from C_v diverges into various paths. Let's assume there are 10 divergent paths from path 1 to path 10. We measure the tracking depth of each path by the number of instructions it has visited. Path 1 is shown in the code is the longest path among all. Here we define the statistics measured for each verified checkpoints:

- *Number of paths:* Total number of paths. This shows the divergence of code tracking required for verification. Total 10 paths are tracked in this example.
- *Max depth:* The depth of the longest path. This means the maximum limit of recursion for dependence tracking. In this example path 1 has the longest depth of 6:
- *Average depth:* Average depth of all paths. In this example, the average depth of 10 paths is 6.
- *Total instructions visited:* This is the total number of instructions visited for all the recursion. This can be used to measure the execution time required for the recursion and also an approximated estimation of the recovery slice code size.

For each checkpoint, we measure these 4 statistics and computed average of each value for checkpoints in a kernel. Note that in the optimal pruning algorithm unnecessary dependence tracking paths are skipped (*short-circuited*), i.e. if a validation state is unsafe for a certain path a PDDG node depends on, other dependences can be ignored. This significantly reduces the portion of the dependence tracking tree the verification must be visited.

Figure 5.10 shows the average number of paths for each checkpoints. This indicates how broad the verification must recursively track the dependence. On average, 3.0 paths must be verified for each checkpoint. Kernel named GPU_laplace3d in the LPS application had the maximum 68 number of paths. Without short-circuit optimization, the checkpoints have 29.0 paths on average and 849 in maximum. This shows that the pruning algorithm efficiently avoids unnecessary visits of dependence paths on verification.

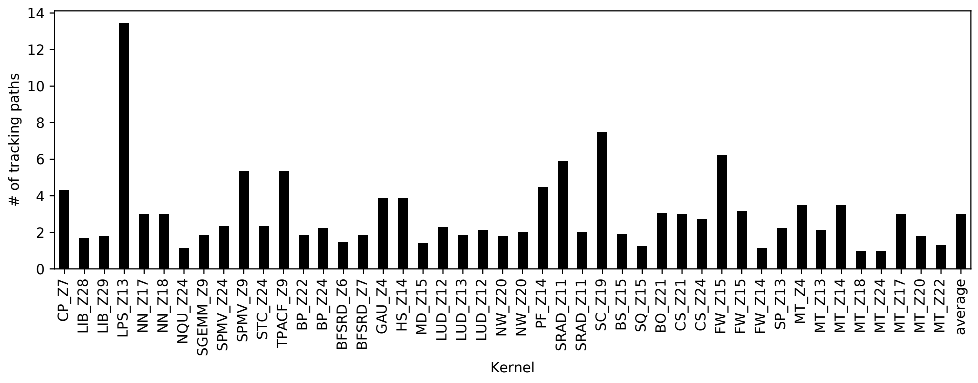


Figure 5.10: Average number of dependence tracking paths for each checkpoint.

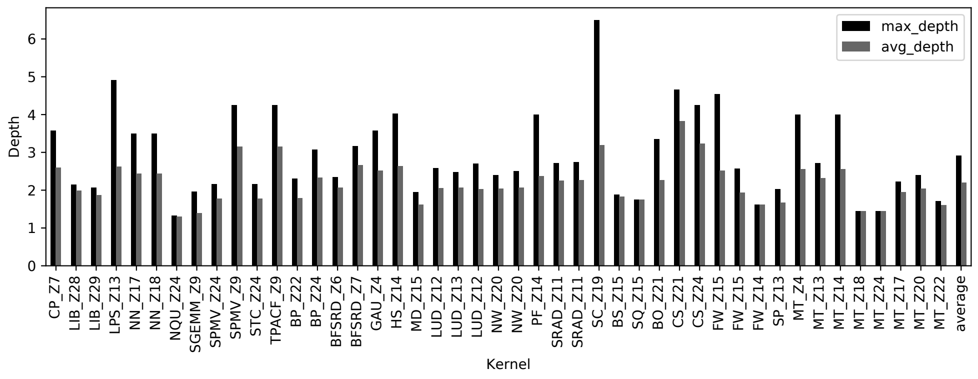


Figure 5.11: Maximum depth and average depth of instruction visited while pruning.

Figure 5.11 presents the maximum and average depth of the dependence tracking. The total average for all kernels is 2.9 for maximum depth and 2.2 for average depth. The checkpoint in executeSecondLayer kernel in SGEMM has the maximum depth of 21. The original tracking path without short-circuiting has an average depth of 3.4 and a maximum depth of 36.

Figure 5.11 shows the total number of instructions visited in the dependence tracking tree. This could be used as an estimated measure of the pruning execution time or the generated size of the recovery slice. On average 6.2 instructions are visited for verification per checkpoint and maximum of 188. Without short-circuit optimization,

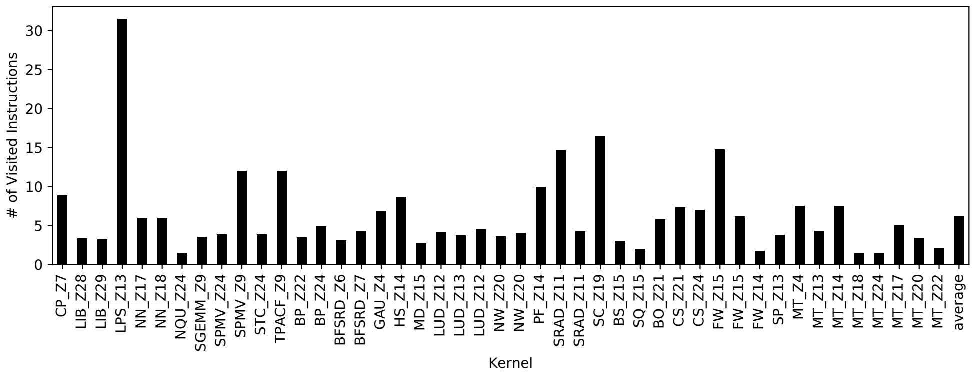


Figure 5.12: Average number of instructions visited for checkpoints while pruning.

this would be 76.6 on average and a maximum of 2393.

Breakdown of Checkpoint Pruning Result

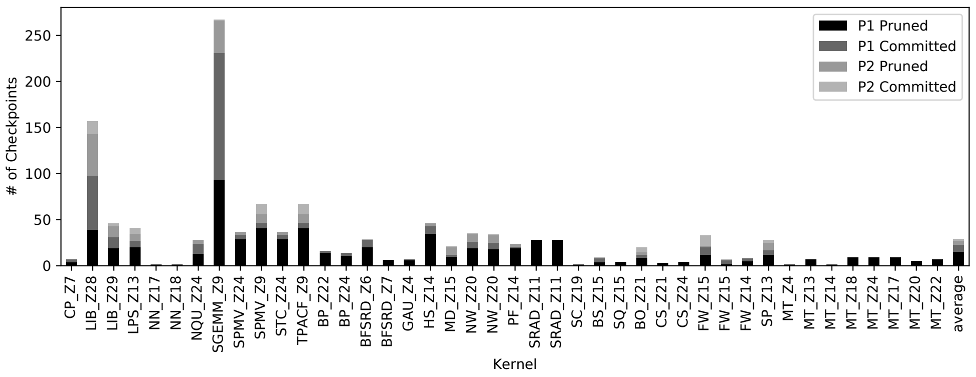


Figure 5.13: Number of checkpoint decision finalized in each phase.

Penny's optimal pruning algorithm efficiently finalizes most of the checkpoints in linear time, and the majority of them are finalized after the first phase which requires less computation compared to the second phase. Figure 5.13 and Figure 5.14 shows the number of checkpoint decisions finalized in each stage and their relative portions. P1 Pruned and P1 Committed are the checkpoints finalized as pruned or committed

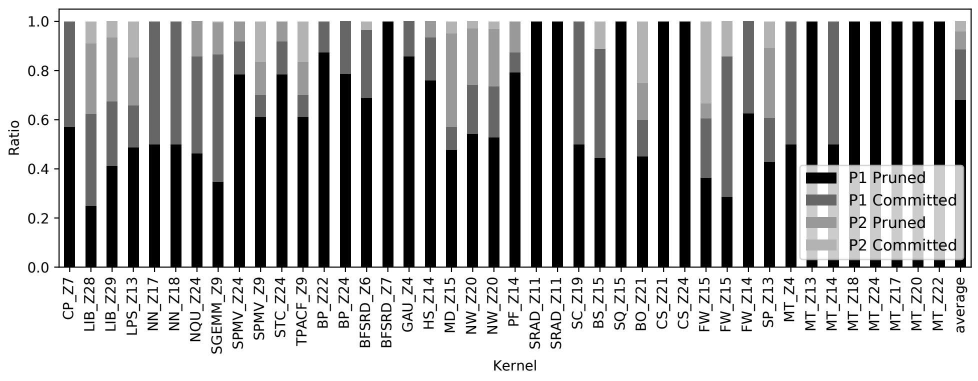


Figure 5.14: Relative portion of checkpoints decision finalized in each phase.

at the first phase. Remaining checkpoints are decided to pruned or committed in the second phase which is presented as P2 Pruned and P2 Committed. On average 88.6% of the checkpoints are finalized to either state after phase 1, thus an only a small portion of checkpoints requires the time-consuming phase 2.

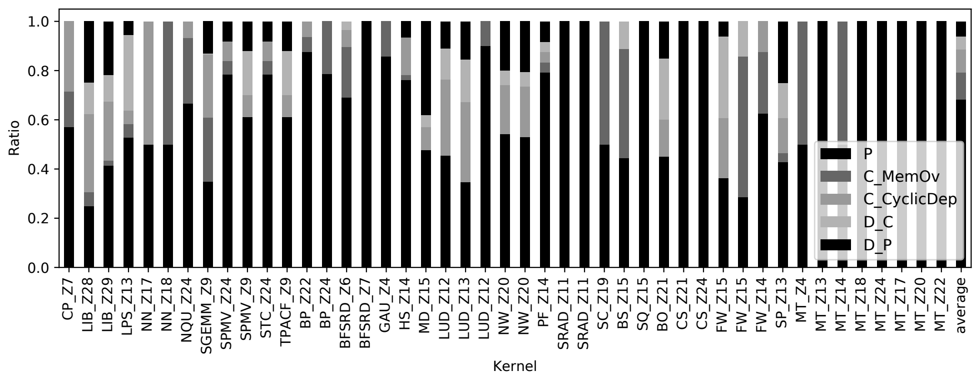


Figure 5.15: Detailed breakdown of pruning decision.

Figure 5.15 shows the detailed breakdown of how checkpoints are classified on each pruning phase. P is the portion of checkpoints being pruned at the first phase. 68.8% of the checkpoints are pruned on average. C_MemOv is the checkpoint being committed after the first phase due to violation of memory overwrite verification, which is 11.1% on average. C_CyclicDep is the portion committed after the fist phase due to cyclic dependence in the verification dependence tracking, accounting for 9.3% on average. For the remaining checkpoints, the second phase collects decision dependence and orders the verification. After the second phase, 5.3% are committed (D_C) and 6.0% is pruned (D_P).

Portion of Checkpoints Eliminated

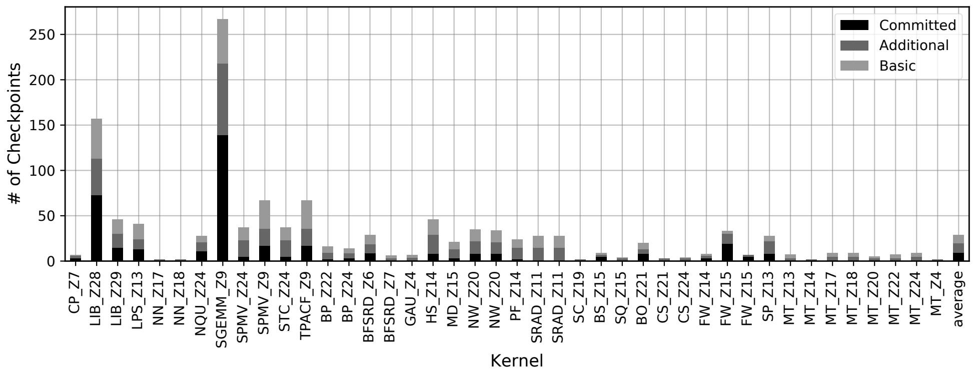


Figure 5.16: Number of checkpoints removed by basic/optimal pruning.

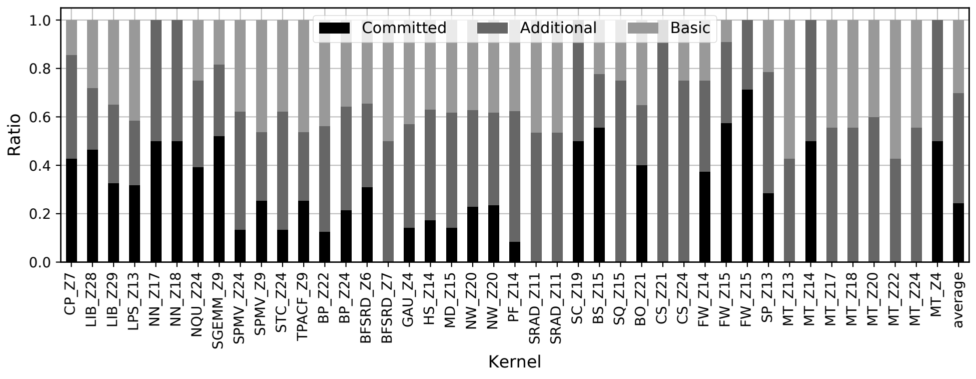


Figure 5.17: Relative portion of checkpoints removed by basic/optimal pruning.

This section studies the statistics of our optimal checkpoint pruning—that can significantly reduce Penny's run-time overhead, as shown in Section 6.2.2 in comparison to Bolt's naive pruning. Figure 5.16 compares the number of checkpoints pruned using each pruning algorithm and Figure 5.17 is the relative portion of the pruned checkpoints normalized to total checkpoints. (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.

5.6 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.

Only the registers that have one or more committing checkpoint remaining after pruning, have storage unit assigned. If a storage alternation is applied to a register and checkpoints colored with both storages remain, the register gets two units of storage assigned. For other checkpoints, a single unit of storage is assigned.

For checkpoint storages, Penny uses shared memory (in SRAM) and global memory (in DRAM but cached) that are both protected by ECC in GPUs [65]. 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* [65]. 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 5.2). 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 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.

Penny also provides an option to use free occupancy-preserving registers for checkpoint storage. With a sacrifice of some resilience, checkpointing cost can be mitigated by the low-latency storage. A DUE may happen when a program register and its corresponding duplicate are corrupted simultaneously, but the chance is inconspicuous.

5.7 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. So, the checkpoint storage address for shared memory can be computed as:

$$\begin{aligned} \mathit{checkpoint\_address} = {} & \underbrace{\mathit{storage\_base} + \mathit{thread\_id} \times \mathit{reg\_size}}_{\text{reused thread\_base}} \\ & + \underbrace{\mathit{storage\_index} \times \mathit{reg\_size} \times \mathit{threads\_per\_SM}}_{\text{computed at compile time}} \end{aligned}$$

The computation is split in two parts. The first $\mathit{storage\_base}$ can be computed once for each thread and reused for all checkpoints. So, the register that the result is stored is remembered and reused if possible. The second part should be computed individually for each checkpoint, because it requires a storage index that is assigned for each checkpointed unit. However, this can be done in compilation time, so in the optimal case, the final address can be calculated in just one addition. If the computation for $\mathit{storage\_base}$ is inside a loop, Penny tries to hoist it out of the loop by LICM if there is a free register throughout the loop to store the computed value.

Finally, Penny performs local checkpoint scheduling to improve the decision made by the bimodal checkpoint scheduling (Section 5.4). 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.

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                                  ▷ Cyclic dependence found
8:         return $\Phi(s) \leftarrow \phi_I$
9:     if $s$ is a constant value then
10:        return $\Phi(s) \leftarrow \phi_V$
11:    if $s$ is a load from read/write memory then
12:        return $\Phi(s) \leftarrow$ CHECKMEMOW($s$, $c_v$)
13:    $\phi_{merged} \leftarrow \phi_V$                    ▷ Initialize validation state before merging
14:    $D \leftarrow$ GETPREDDATADEPS($s$)                  ▷ For all predicate/data dependences
15:    for $\forall s_d \in D$ do
16:        $\phi_{dep} \leftarrow$ MARKING($c_v$, $Visited \cup \{s\}$, $s_d$)
17:        $\phi_{merged} \leftarrow$ MAXPRIORITY($\phi_{merged}$, $\phi_{dep}$)      ▷ Merge validation states
18:    if $\phi_{merged} = \phi_I$ and $s \in \mathbb{C}$ then            ▷ $\mathbb{C}$: set of all checkpoints
19:        $\phi_{merged} \leftarrow \phi_U$
20:    return $\Phi(s) \leftarrow \phi_{merged}$
21: function GETPREDDATADEPS($s$) ▷ $s$ Collect dependences on control flow graph
22:    $D_{data} \leftarrow \{s_d | s \xrightarrow{data} s_d\}$            ▷ $s$ has a data dependence on $s_d$
23:    $D_{pred} \leftarrow \{s_p | s \xrightarrow{pred} s_p\}$       ▷ $s$ has a predicate dependence on $s_d$
24:    return $D_{data} \cup D_{pred}$
```

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 regions
   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
7:         return $\emptyset$                              ▷ Stop if a cyclic dependence is found
8:     if $s \in \mathbb{C}$ and $T(s) = \tau_C$ then              ▷ For a committed ($\tau_C$ ) checkpoint
9:         return OWCKPTS($s$, $Exp_{end}(c_v)$)
10:    $F \leftarrow \emptyset$                          ▷ Set of nodes $c_v$ has decision dependences on
11:    if $s \in \mathbb{C}$ and $T(s) = \tau_U$ then              ▷ For an undecided ($\tau_U$) checkpoint
12:        $F \leftarrow F \cup \{s\} \cup$ OWCKPTS($s$, $Exp_{end}(c_v)$)
13:    $D \leftarrow$ GETPREDDATADEPS($s$)                           ▷ From Algorithm 1
14:    for $\forall s_d \in D$ do
15:        $F \leftarrow F \cup$ GETDECISIONDEPS($c_v$, $Visited \cup \{s\}$, $s_d$)
16:    return $F$
```

Chapter 6

Evaluation

6.1 Test Environment

6.1.1 GPU Architecture and Simulation Setup

Model	Nvidia Tesla C2050
SM Count	14
Shading Units	448
Register	512KB / SM
L1 Cache	16KB/SM
Shared Mem	48KB/SM
L2 Cache	768 KB
Device Memory	3GB

Table 6.1: Specification of the simulated GPU.

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 [11] use PTX code as a basis for the cycle-level simulation, and tools such as CRAT [88] 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. Detailed specification of the GPU is in Table 6.1. Note that the L1 cache is interchangeable with shared memory, so the L1 cache can be increased by configuration when shared memory is not used.

6.1.2 Tested Applications

Suite	Application	Abbr.	Suite	Application	Abbr.
GPGPU-Sim bench [11]	Coulombic potential	CP	Parboil	2-point angular correlation	TPACF
	Libor Monte Carlo	LIB		SP Matrix multiplication	SGEMM
	Laplace transform	LPS	Rodinia [19]	Back propagation	BP
	Neural network	NN		Breadth-first search	BFS
	N Queen	NQU		Gaussian Elimination	GAU
CUDA toolkit samples [67]	Binomial options	BO		Hotspot	HS
	Black-Scholes	BS		Molecular Dynamics	MD
	Convolution separable	CS		Needleman-Wunsch	NW
	Scalar product	SP		Pathfinder	PF
	Sobol filter	SQ		Speckle reducing anisotropic diffusion	SRAD
	Fast Walsh transform	FW		stream cluster	SC
	Matrix transpose	MT			
Parboil [74]	Sparse matrix-vector mult.	SPMV			
	Jacobi stencil	STC			

Table 6.2: Applications used for evaluation.

We used various CUDA applications from multiple benchmark suites: the benchmarks included in the GPGPU-Sim [11], sample codes bundled with CUDA toolkit [67], Parboil benchmark suite [74], and Rodinia benchmark suite [19]. Table 6.2 shows benchmark applications used in our simulations. Some applications were not able to be executed due to: 1) CUDA version incompatibility between the original version the application is targeting and the version supported by GPGPU-sim. 2) Unimplemented CUDA features that are less frequently used such as special instructions to access texture memory. And from the applications that were possible to be executed, we excluded the applications with no memory anti-dependence, i.e. the kernel can be idempotently re-executed from the beginning. In this case, there are no execution overheads compared to original code, so we excluded them for a more fair comparison.

6.1.3 Register Assignment

In a GPGPU program, the number of registers each thread uses is one of the decision factors of GPU occupancy. Decreasing the number may cause register spills into global memory, but more threads can be concurrently run, and increasing it has an opposite effect up to a certain point. Most of the known GPU programming system compilers use the minimum register number that does induce spill, and this is commonly used for programming.

A user can specify the number of registers to the compiler, but this is hard to decide other than profiling it empirically, and it does not translate across different hardwares or other compiler setting changes.

So we follow the common practice of choosing the minimum number of registers not spilling in our experiments. As a result, the evaluation results in the following number of registers used for each thread and the number of blocks assigned for each SM can be different between different settings.

6.2 Performance Evaluation

6.2.1 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 3.4). The following schemes are tested.

- **iGPU** This is De Kruijf et al. [29]'s iGPU [60] 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 [53] 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.

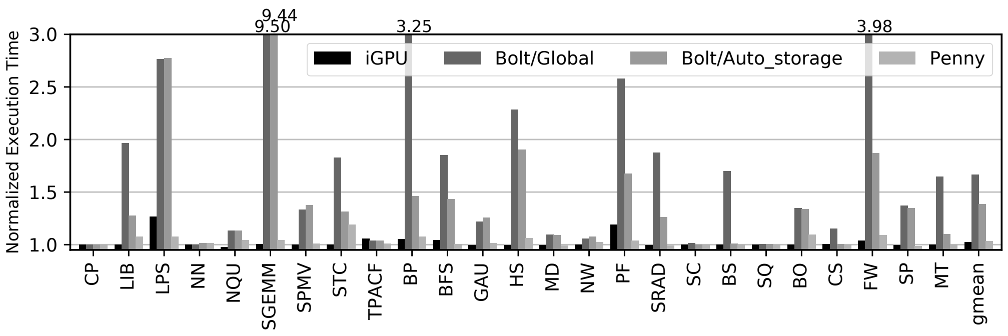


Figure 6.1: Fault-free execution time overhead.

Figure 6.1 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 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 dependencies are originated from program semantics that prohibits Penny's checkpoint pruning and bimodal checkpoint placement.

6.2.2 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 6.2 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 *automatic storage assignment optimization (ASAO)*, the overhead of which is represented in the

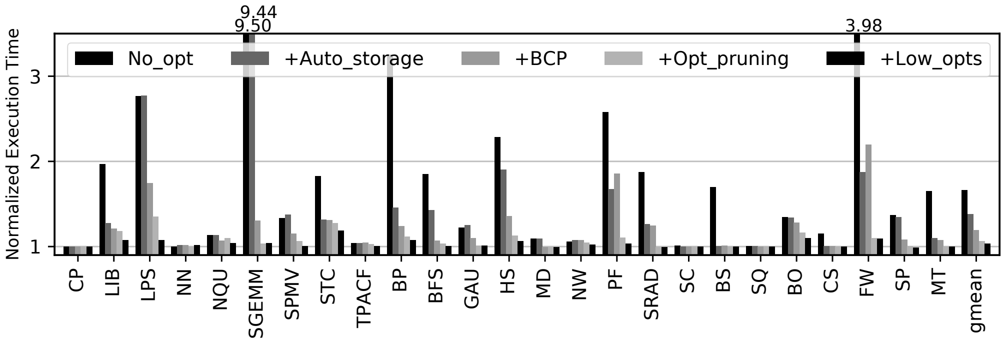


Figure 6.2: Impact of Penny optimizations accumulated.

+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 5.7) 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.

6.2.3 Assigning Checkpoint Storage and Its Integrity

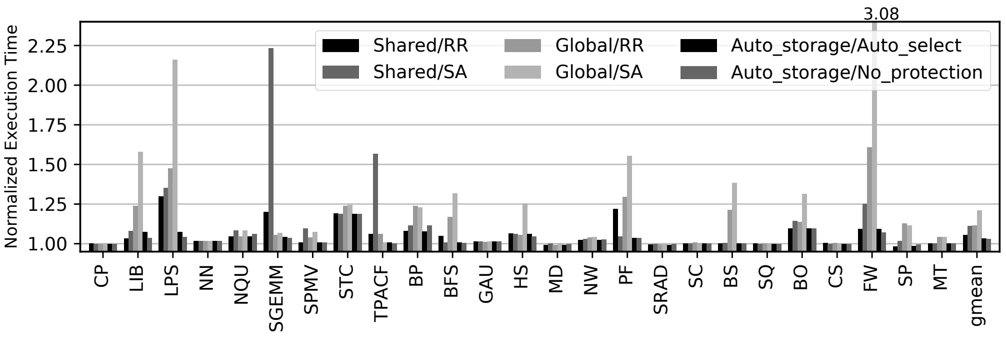


Figure 6.3: Storage assignment and overwrite prevention.

This section provides sensitivity analysis results on different checkpoint storage assignment schemes and checkpoint overwriting prevention schemes. In Figure 6.3, 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.

6.2.4 Impact of Optimal Checkpoint Pruning

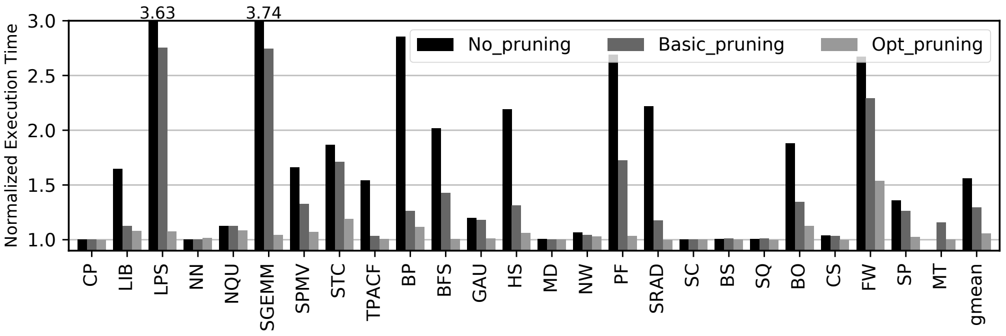


Figure 6.4: Performance impact of basic/optimal pruning.

In Section 5.5.4 we have shown the effect of basic and optimal pruning by analyzing the compiler statistics. Here we evaluate how the eliminated checkpoints translate to the run-time overhead reduction. As shown in Figure 6.4, 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.

6.2.5 Impact of Alias Analysis

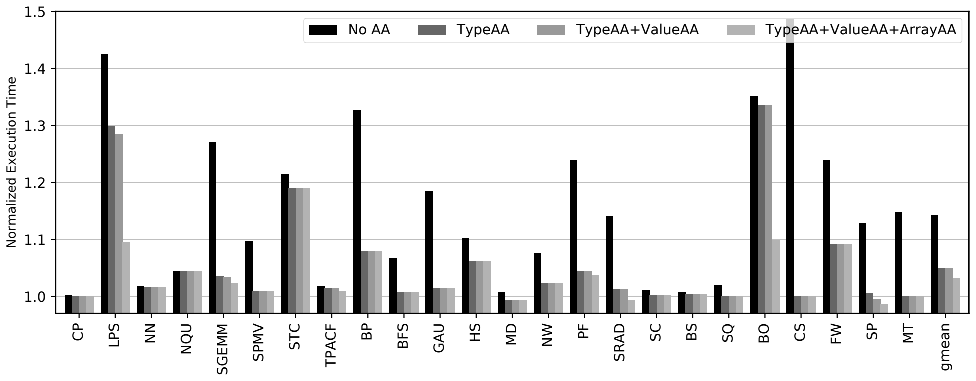


Figure 6.5: Performance impact of various alias analysis.

Penny uses alias analysis to identify alias in memory addresses. The analysis result is mainly used in two transformation phases: 1) in the region formation phase, the region formation algorithm find maximal regions that separate memory anti-dependences and 2) in the checkpoint pruning phase, memory overwrite is tested with the alias result. Even with the simplest alias analysis, Penny guarantees safe recovery by conservatively handling may-aliases though the performance may not be optimal due to the false-positive aliases.

We implemented a few alias analysis algorithms for the GPU. Since CUDA does not have general pointer variables, simple alias analysis schemes were efficient enough for most cases. Here is the explanation of the analysis schemes we have used:

- No AA: No alias analysis scheme used and all addresses are identified as an alias.

- TypeAA: Simple type-based alias analysis. Compare the instruction operand type specifier.

- ValueAA: If the constant offset matches and the register value is unchanged between two points return must alias.
- ArrayAA: Backtrack the address value and find where the base value of the address (e.g. base of an array) comes from. If the base does not match, two addresses are no alias.

Figure 6.4 shows the result of using different combinations of aliases. The first setting does not use any alias analysis and the following settings turn on the alias analysis schemes one by one. Note that using multiple schemes gives a more accurate result because the analysis results from each scheme complement each other. Not using any alias analysis (No AA) gives 14.3% of overhead on average. Applications including LPS, SGEMM, BP, PF, CS, and FW shows significant slowdown. Using simple type-based analysis (TypeAA) significantly improves performance in many cases reducing the average overhead to 5.0%. Adding the value-based analysis slightly reduces the average overhead to 4.9%. Array base analysis shows a noticeable improvement in applications such as LPS and BO. The average overhead is reduced to 3.1%.

6.3 Repurposing the Saved ECC Area

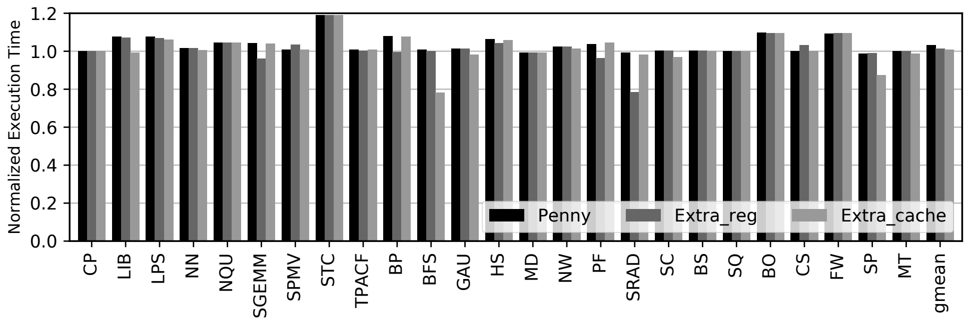


Figure 6.6: Repurposing the saved HW area.

Figure 6.6 is the performance result of repurposing the save area by replacing

SECDED ECC into single-bit parity for Penny, into additional RFs or L1 cache/shared memory (L1 and shared memories are interchangeable by configuration in Fermi GPUs). Penny is the execution of Penny without adding additional HW components. Extra_reg is the result of repurposing the saved area into additional RF. Our estimation based on the synthesis result shows that 93.1kB of registers can be added to the original 512kB/SM of Tesla C2050, by using the saved area. This area can also be translated into 89.6kB of L1 cache/shared memory. We add additional 16kB/73.6kB to the original 16kB/48kB of L1 cache/shared memory for Extra_cache. SGEMM and SRAD show noticeable performance improvement for adding extra registers reducing the execution time up to 21.5% compared to non-checkpointed execution. For extra cache, BFS and SP show improvement, reducing the execution up to 22.8%.

6.4 Energy Impact on Execution

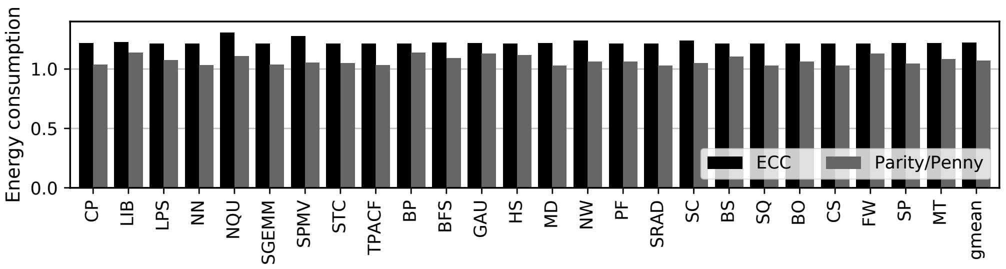


Figure 6.7: Energy consumption of RF.

In addition to the hardware synthesis (Section 2.2), 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 2.2 and Table 2.3 to GPGPU-Sim's power simulator, i.e., GPUWatch[46]. Figure 6.7 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.

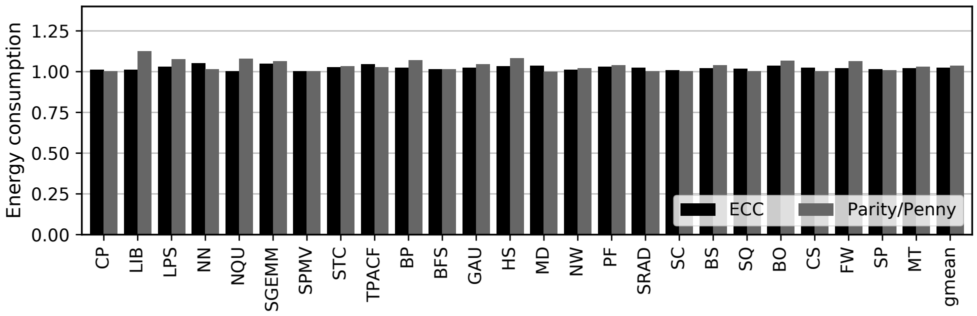


Figure 6.8: Energy consumption of GPU.

Figure 6.8 presents the energy consumed by entire GPU. Though RF is one of the most energy-consuming modules on the core, the portion is not dominant among all components in the chip, and also off-chip modules such as device memory take up a large portion of GPU energy consumption. Thus, the RF energy savings has a minor impact on the total energy consumption, while the additional energy used by the extended execution time generally renders a significant effect. In applications such as in LIB the execution time if increased for checkpointing, so the energy consumption is increased. The ECC protected GPU consumed 2.4% of more energy wile Penny used 3.7% more.

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 which will conduct more design space exploration and performance optimization to fully realize the benefits. Apart from that, it is still critical to reducing RF energy itself. The reason is that a register file (RF) determines the GPU's nominal voltage (Vdd) which must be set high enough to handle the worse-case voltage demand [47]. 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. Additional discussions on the total GPU energy consumption are deferred to Section 8.1.

6.5 Performance Overhead on Volta Architecture

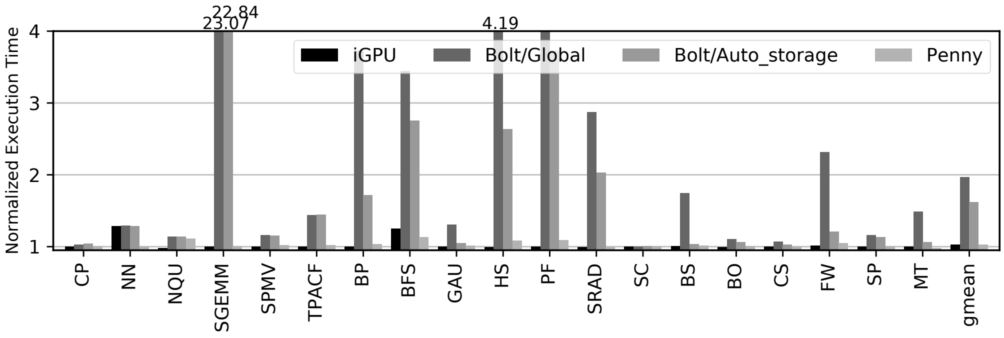


Figure 6.9: Performance comparison on Titan V.

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 6.9 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 6.1). Overall, the run-time overhead of Penny is only 3.6% on average.

6.6 Compilation Time

The translation algorithm suggested Penny is also efficient in terms of compilation time. Figure 6.10 shows the compilation time spent for each kernel broken down into region formation, checkpoint coloring, checkpoint pruning, code generation, and others. Intel

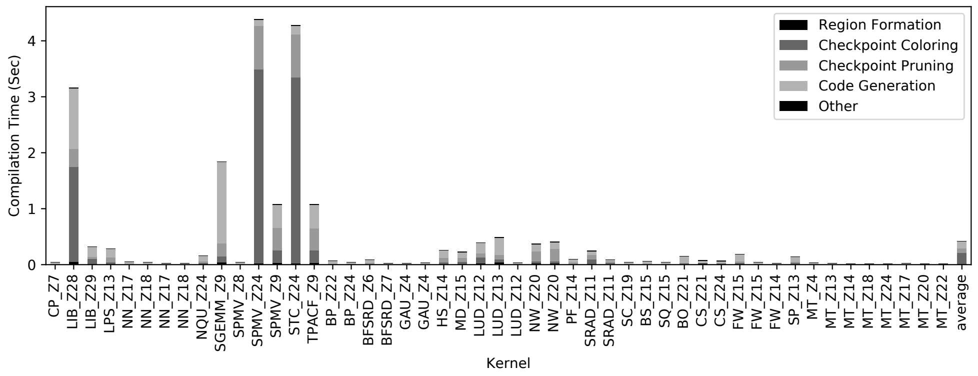


Figure 6.10: Compilation time of Penny.

Core i7-9700 CPU was used for compilation. It shows that the average of the total compilation time is 425 milliseconds and 4.39 seconds at maximum. The average time spent for region formation, checkpoint coloring, checkpoint pruning, code generation, and others is 11, 198, 82, 128, and 6 milliseconds respectively.

Note that this result is a compilation time measured for a single full compilation. The automatic optimization module may produce multiple versions of code to compare and select the best optimization policy, thus the compilation can be repeated a few times.

Chapter 7

Related Works

Over the years, many researchers have leveraged *idempotence* for various purposes. Mahlke et al. were the first to exploit the idea, which they used to recover from exceptions during speculative execution in a VLIW processor [57]. Around the same time, Bershad et al. proposed *restartable atomic sequences* for a uniprocessor based on idempotence [14]. Kim et al. leveraged idempotence to reduce the hardware storage required to buffer data in their compiler-assisted speculative execution model [43]. Hampton et al. used idempotence to support fast and precise exceptions in a vector processor with virtual memory [38]. Tseng et al. used idempotent regions for data-triggered thread execution [82]. Since then, it has been used for various applications, such as reducing the speculative storage overflow [43], supporting exceptions in a vector processor with virtual memory [38] and data-triggered thread execution [82].

Recently, researchers have leveraged idempotence for recovery from soft errors [35, 29]. Also, Liu et al. [53] 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. [54, 55, 52] 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 [20, 21] have started using idempotent processing to recover from the frequent power failures that occur in systems without batteries [87, 83, 51].

Xie et al. [87] use idempotence-based recovery and heuristics to approximate minimal checkpoints (logs) to survive power failures. Their design revolves around the idea of severing anti-dependences by placing a checkpoint between a load-store pair, in a manner reminiscent of Feng et al. [35] and de Kruijf et al. [29]. Lately, their techniques were used by Woude et al. [83] to highlight both the promise and the limitations of using idempotence to ensure forward progress when multiple power failures occur within a span of microseconds. In a similar vein, Liu et al. [51] highlight the limitations of anti-dependence based idempotence analysis in terms of additional power consumption due to unnecessary checkpoints. 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 [33, 49, 34, 64]. The most closely-related work is iGPU that leverages idempotent recovery for exception handling, context switching, and timing speculation [60]. 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.

Chapter 8

Conclusion and Future Works

Given the large GPU register file (RF) size and the ever-growing trend, protecting RFs with ECC at the cost of increasing hardware complexity and power consumption poses significant challenges for GPU architects. Furthermore, near-threshold-voltage computing systems should be able to handle wider-cardinality multi-bit errors, which requires more expensive ECC protection.

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 $\approx$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.

8.1 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 [47]. 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.

Bibliography

[1] Nvidia tesla v100 gpu architecture. Technical report, Nvidia, 2017.

[2] Nvidia truring gpu architecture. Technical report, Nvidia, 2018.

[3] F. Alzahrani and T. Chen. On-chip tec-qed ecc for ultra-large, single-chip memory systems. In *ICCD'94*. IEEE, 1994.

[4] Saman Amarasinghe, Dan Campbell, William Carlson, Andrew Chien, William Dally, Elmootazbellah Elnohazy, Robert Harrison, William Harrod, Jon Hiller, Sherman Karp, Charles Koelbel, David Koester, Peter Kogge, John Levesque, Daniel Reed, Robert Schreiber, Mark Richards, Al Scarpelli, John Shalf, Allan Snavely, and Thomas Sterling. Exascale software study: Software challenges in extreme scale systems, 2009.

[5] Saman Amarasinghe, Mary Hall, Richard Lethin, Keshav Pingali, Dan Quinlan, Vivek Sarkar, John Shalf, Robert Lucas, Katherine Yelick, Pavan Balanji, Pedro C. Diniz, Alice Koniges, and Marc Snir. Exascale programming challenges. In *Proceedings of the Workshop on Exascale Programming Challenges, Marina del Rey, CA, USA*. U.S Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), Jul 2011.

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

[7] Jim Ang, Brian Carnes, Patrick Chiang, Doug Doerfler, Sudip Dosanjh, Parks Fields, Ken Koch, Jim Laros, Matt Leininger, John Noe, Terri Quinn, Josep Torrellas, Jeff Vetter, Cheryl Wampler, and Andy White. Exascale hardware architectures working group. Technical report, Lawrence Livermore National Laboratory, 2011.

[8] ARM. Developer suite, 2003. Version 1.2.

[9] Todd Austin and Valeria Bertacco. Deployment of better than worst-case design: Solutions and needs. In *ICCD'05*, 2005.

[10] Todd Austin, Valeria Bertacco, David Blaauw, and Trevor Mudge. Opportunities and challenges for better than worst-case design. In *ASP-DAC'05*, 2005.

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

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

[13] Salomon Beer, Marco Cannizzaro, Jordi Cortadella, Ran Ginosar, and Luciano Lavagno. Metastability in better-than-worst-case designs. In *ASYNC'14*, 2014.

[14] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *ASPLOS V*, 1992.

[15] Robin M Betz, Nathan A DeBardeleben, and Ross C Walker. An investigation of the effects of hard and soft errors on graphics processing unit-accelerated molecular dynamics simulations. *Concurrency and Computation: Practice and Experience*, 26(13):2134–2140, 2014.

[16] Shekhar Borka. The exascale challenge. In *International Symposium on VLSI Design Automation and Test*, 2010.

[17] Shekhar Borkar. Exascale computer-a fact or a fiction. *Keynote address: IEEE International Parallel and Distributed Processing Symposium*, 2013.

[18] Preston Briggs, Keith D Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, 1994.

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

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

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

[22] Cristian Constantinescu. Trends and challenges in vlsi circuit reliability. In *MICRO 36*, 2003.

[23] W.J. Cook, W.H. Cunningham, W.R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. Wiley Series in Discrete Mathematics and Optimization. Wiley, 2011.

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

[25] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *ISCA'10*, 2010.

[26] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent processor architecture. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 140–151. ACM, 2011.

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

[28] Marc A. De Kruijf. *Compiler Construction of Idempotent Regions and Applications in Architecture Design*. PhD thesis, Madison, WI, USA, 2012.

[29] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *PLDI 2012*, 2012.

[30] Peng Du, Piotr Luszczek, and Jack Dongarra. High performance dense linear system solver with soft error resilience. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 272–280. IEEE, 2011.

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

[32] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA'11*, 2011.

[33] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications. In *ISPASS'14*. IEEE, 2014.

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

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

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

[37] Jiong Guo, Falk Hüffner, Erhan Kenar, Rolf Niedermeier, and Johannes Uhlmann. Complexity and exact algorithms for vertex multicut in interval and bounded treewidth graphs. *European Journal of Operational Research*, 186(2):542–553, 2008.

[38] Mark Hampton and Krste Asanović. Implementing virtual memory in a vector processor with software restart markers. In *ICS 2006*, 2006.

[39] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. In *MICRO 31*, 2011.

[40] Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *DAC 2013*, 2013.

[41] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. Near-threshold voltage (ntv) design: Opportunities and challenges. In *DAC'12*, 2012.

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

[43] Seon Wook Kim, Chong-Liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Exploiting reference idempotency to reduce speculative storage overflow. In *TOPLAS'06*, 2006.

[44] Denés Konig. Gráfok és mátrixok. matematikai és fizikai lapok, 1931.

[45] Lingbo Kou. Impact of process variations on soft error sensitivity of 32-nm vlsi circuits in near-threshold region. Master's thesis, 2014.

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

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

[48] Jingwen Leng, Yazhou Zu, Minsoo Rhu, Meeta Gupta, and Vijay Janapa Reddi. Gpuvolt: Modeling and characterizing voltage noise in gpu architectures. In *ISLPED'14*, 2014.

[49] Guanpeng Li, Karthik Pattabiraman, Chen-Yong Cher, and Pradip Bose. Understanding error propagation in gpgpu applications. In *SC'16*, 2016.

[50] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *MICRO 51*, 2018.

[51] Qingrui Liu and Changhee Jung. Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems. In *NVMSA 2016*, 2016.

[52] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. Clover: Compiler directed lightweight soft error resilience. In *LCTES'15*, 2015.

[53] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In *SC'16*, Nov 2016.

[54] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. Compiler directed soft error detection and recovery to avoid due and sdc via tail-dmr. *TECS'16*, 2016.

[55] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. Low-cost soft error resilience with unified data verification and fine-grained recovery. In *MICRO 49*, Oct 2016.

[56] Robert Lucas, James Ang, Keren Bergman, Shekhar Borkar, William Carlson, Laura Carrington, George Chiu, Robert Colwell, William Dally, Jack Dongarra, Al Geist, Gary Grider, Rud Haring, Jeffrey Hittinger, Adolfy Hoisie, Dean Klein, Peter Kogge, Richard Lethin, Vivek Sarkar, Robert Schreiber, John Shalf, Thomas Sterling, and Rick Stevens. Top ten exascale research challenges. Technical report, U.S. Department of Energy ASCAC Subcommittee, Boston, MA, USA, Feburary 2014.

[57] Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. Sentinel scheduling for vliw and superscalar processors. In *ASPLOS V*, 1992.

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

[59] Gokhan Memik, Mahmut T. Kandemir, and Ozcan Ozturk. Increasing register file immunity to transient errors. In *DATE*, 2005.

[60] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. igpu: Exception support and speculative execution on gpus. In *ISCA'12*, 2012.

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

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

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

[64] Bin Nie, Lishan Yang, Adwait Jog, and Evgenia Smirni. Fault site pruning for practical reliability analysis of gpgpu applications. In *MICRO 51*. IEEE, 2018.

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

[66] NVIDIA. Cuda toolkit 3.2 math library performance, 2010. http://developer.download.nvidia.com/compute/cuda/3_2/docs/CUDA_3.2_Math_Libraries_Perform

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

[68] Robert Pawlowski. *Measurement and Analysis of Soft Error Vulnerability of Low-Voltage Logic and Memory Circuits*. PhD thesis, Corvallis, OR, USA, 2015.

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

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

[71] Lattice Semiconductor. Lattice ecc module reference design, 2018. http://www.latticesemi.com.

[72] Muhammad Shafique, Siddharth Garg, Jörg Henkel, and Diana Marculescu. The eda challenges in the dark silicon era: Temperature, reliability, and variability perspectives. In *DAC '14*, 2014.

[73] M. Snir, R. W. Wisniewski, J. A. Abraham, V. Adve, S. Bagchi, P. Balaji, J. Belak, F. Cappello P. Bose, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeieben, P. Diniz, M. Erez C. Engelmann, S. Fazzari, A. Geist, R. Gupta, F. Johnson, Krishnamoorthy, S. Leyffer, T. Munson D. Liberty, Mitra, R. Schreiber, J. Stearley, and E. V. Hensbergen. Addressing failures in exascale computing. *"International Journal of High Performance Computing Applications"*, 28(2), 2014.

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

[75] Synopsys. Compiler, design and user, rtl and guide, modeling. 2001. http://www. synopsys. com.

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

[77] Michael B. Taylor. Is dark silicon useful?: Harnessing the four horsemen of the coming dark silicon apocalypse. In *DAC'12*, 2012.

[78] 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. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. In *HPCA '15*. IEEE, 2015.

[79] Devesh Tiwari, Saurabh Gupta, George Gallarno, Jim Rogers, and Don Maxwell. Reliability lessons learned from GPU experience with the titan supercomputer

at oak ridge leadership computing facility. In Jackie Kern and Jeffrey S. Vetter, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 38:1–38:12. ACM, 2015.

[80] Josep Torrellas, Daniel Quinlan, Allan Snavely, and Wilfred Pinfold. Thrifty: An exascale architecture for energy-proportional computing, 2013.

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

[82] Hung-Wei Tseng and Dean M Tullsen. Cdtt: Compiler-generated data-triggered threads. In *HPCA'14*, 2014.

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

[84] Ross C Walker and Robin M Betz. An investigation of the effects of error correcting code on gpu-accelerated molecular dynamics simulations. In *Proceedings of the conference on extreme science and engineering discovery environment: Gateway to discovery*, page 8. ACM, 2013.

[85] Liang Wang and Kevin Skadron. Implications of the power wall: Dim cores and reconfigurable logic. *IEEE Micro*, pages 40–48, 2013.

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

[87] Mimi Xie, Mengying Zhao, Chao Pan, Jingtong Hu, Yongpan Liu, and Chun Xue. Fixing the broken time machine: Consistency-aware checkpointing for energy harvesting powered non-volatile processor. In *DAC'15*, 2015.

[88] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. CRAT: enabling coordinated register allocation and thread-level parallelism optimization for gpus. *TC'08*, 2018.

[89] Doe Hyun Yoon and Mattan Erez. Memory mapped ecc: Low-cost error protection for last level caches. In *ISCA'09*, 2009.

[90] Wei Zhang, Marc de Kruijf, Ang Li, Shan Lu, and Karthikeyan Sankaralingam. Conair: Featherweight concurrency bug recovery via single-threaded idempotent execution. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 113–126, 2013.

초 록

반도체 미세공정 기술이 발전하고 문턱전압 근처 컴퓨팅(near-threashold voltage computing)이 도입됨에 따라서 소프트 에러로부터의 복원이 중요한 과제가 되었다. 강력한 병렬 계산 성능을 지닌 GPU는 고성능 컴퓨팅에서 중요한 위치를 차지하게 되었고, 슈퍼 컴퓨터에서 쓰이는 GPU들은 에러 복원 코드인 ECC를 사용하여 레지스터 파일 및 메모리 등에 저장된 데이터를 보호하게 되었다. 하지만 레지스터 파일에 ECC를 사용하는 것은 큰 하드웨어나 에너지 비용을 필요로 한다.

이런 값비싼 ECC의 하드웨어 비용을 줄이기 위해 본 논문에서는 컴파일러 기반의 저비용 GPU 레지스터 파일 복원 기법인 Penny를 제안한다. 이는 최신의 멱등성(idempotency) 기반 에러 복원 기법을 저비용의 에러 검출 코드(EDC)와 결합한 것이다. 본 논문은 다음 두가지 문제를 해결하는 데에 집중한다.

1. *에러 검출 코드 기반으로 멱등성 기반 에러 복원을 사용시 소프트 에러로부터의 안전한 복원을 보장할 수 있는가?* 본 논문에서는 에러 검출 코드가 멱등성 기반 복원 기술과 같이 사용되었을 경우 기존의 복원 기법에서 필요로 했던 조건들 없이도 안전하게 에러로부터 복원할 수 있음을 보인다.

2. *체크포인팅에드는 비용을 어떻게 절감할 수 있는가?* GPU는 스토어 버퍼가 없는 등 아키텍쳐적인 특성으로 인해서 CPU와 비교하여 체크포인트 값을 저장하는 데에 큰 오버헤드가 든다. 이 문제를 해결하기 위해 본 논문에서는 다양한 컴파일러 최적화 기법을 통하여 오버헤드를 줄인다.

주요어: GPU, 에러 복원, ECC, idempotence

학번: 2012-30204

Acknowledgements

First I want to thank my advisor Jaejin Lee at Seoul National University for the guidance and patience for me over my graduate school life. I also thank Professor Changhee Jung who had also mentored me during cooperative research. They both had been given guidance that not only helped me grow as a researcher but also as a human.

And I also thank other committee members Professor Soo-Mook Moon, Jin-Soo Kim, and Jae Lee for the support of my research. My research has become more sound and complete with their guidance.

I also thank all the fellow students in the MCRL lab at SNU. I have enjoined exchanging creative thoughts and learned a lot by them. I also appreciate the fellows I have worked with while I was visiting Virginia Tech and Purdue. You guys have given me a lot of help when I was in difficult situations.

Thanks to my family who has always believed in me. My parents have always given me warm support for whatever I am doing. And thanks for my little brother for always being there and giving me good pieces of advice. I also want to thank my new family in Gangneung and San Fransisco. I cannot appreciate more for how much they have welcomed me as a family and I am obliged to return all the support they have given.

Finally, for my best friend and wife Christina, thank you for being with me for all the happy and hard moments. I couldn't have done it without you. We have come through difficult times and I believe the future ahead of us is bright with joy.