ARCHITECTURAL APPROACHES TO DESIGN RELIABLE AND ENERGY-EFFICIENT GPUS

A Dissertation

Presented to

the Faculty of the Department of Electrical and Computer Engineering

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

in Electrical Engineering

by

Jingweijia Tan

May 2016

ARCHITECTURAL APPROACHES TO DESIGN RELIABLE AND ENERGY-EFFICIENT GPUS

Jingweijia Tan

Approved:

Chair of the Committee
Xin Fu, Assistant Professor,
Electrical and Computer Engineering

Committee Members:

Jinghong Chen, Associate Professor,
Electrical and Computer Engineering

Yuhua Chen, Associate Professor,
Electrical and Computer Engineering

Jiming Peng, Associate Professor,
Industrial Engineering

Guoning Chen, Assistant Professor,
Computer Science

Shuaiwen Leon Song, Research Scientist,
Pacific Northwest National Laboratory

Suresh K. Khator, Associate Dean,
Cullen College of Engineering

Badri Roysam, Professor and Chair,
Electrical and Computer Engineering

ARCHITECTURAL APPROACHES TO DESIGN RELIABLE AND ENERGY-EFFICIENT GPUS

An Abstract

of a Dissertation

Presented to

the Faculty of the Department of Electrical and Computer Engineering

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

in Electrical Engineering

By

Jingweijia Tan

May 2016

Abstract

Modern graphic processing units (GPUs) support thousands of concurrent threads and provide high computational throughput, which makes them popular platforms for general-purpose high-performance computing (HPC) applications. However this raises reliability and energy-efficiency challenges in GPU architecture design. Originally designed for graphics applications with relaxed requirements on execution correctness, GPUs lack the error detection and fault tolerance features. In contrast, HPC programs have rigorous demands on execution correctness, which poses serious reliability challenges for general purpose computing on GPUs (GPGPUs). In addition, GPUs consume large amount of energy to achieve its high computing power. The peak power consumption of a high-end GPU is more than twice of the CPU counterparts and the energy-efficiency of GPUs fail to grow as fast as the performance improvement.

In this dissertation, we introduce several architectural approaches to design reliable and energy-efficient GPUs. We first propose several opportunistic techniques to recycle the idle time of streaming processors for soft-error detection and obtain the good fault coverage with negligible performance degradation. Utilizing the promising benefits of resistive memory, we further propose to leverage resistive memory to enhance the soft-error robustness and reduce the power consumption of registers in the GPUs. We then explore to mitigate the susceptibility of GPU register file to process variations. The proposed techniques are able to significantly optimize GPUs' performance under process variations. After that, we propose an effective and low-cost mechanism to maintain the register file reliability with negligible performance loss under process variations and low supply voltages, which enables substantial energy savings via aggressive supply voltage reduction. Finally, we propose an energy-efficient GPU L2 cache design that leverages locality similarity to reduce the L2 energy consumption with negligible performance degradation. Overall, these techniques efficiently address the reliability and energy-efficient challenges in GPU architectures.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The state-of-the-art graphics processing units (GPUs) possess strong computing power and achieve high teraflops peak performance by exploiting massive thread-level parallelism (TLP). Recent enhancement in programming models [1, 2] further motivates researchers to exploit parallelism of a wide range of high-performance computing applications (such as scientific, financial, biological, and graphic applications) on GPUs. Nowadays, GPUs have been widely adopted for the general-purpose computing, known as GPGPUs. However, as process technology keeps scaling down, the increasing reliability and energy-efficient challenges have become growing threats to GPU architecture design.

As one of the major reliability threats, soft errors become severe in GPUs. Soft errors, also called transient faults or single upset events, are failures caused by high-energy neutron or alpha particle strikes in integrated circuits. These faults are said to be "soft" in contrast to hard-faults which are permanent in the device. Soft errors may silently corrupt the data and lead to erroneous computation results. Soft error rate (SER) has been predicted to increase exponentially with the shrinking of feature sizes and growing integration density [3, 4], and GPUs with hundreds of cores integrated in a single chip are prone to suffer severe soft error attacks. For examples, [5] has already observed eight soft errors in a 72-hour run of testing program on 60 NVIDIA GeForce 8800GTS 512. [6] finds that the silent data corruption ratio in commodity GPU is 16-33%, while the ratio is smaller than 2.3% in CPUs which have comparatively developed fault tolerance techniques. The increasing SER becomes the major obstacle to current and future GPU design.

GPU also consumes a large amount of energy to provide high computing capability. For example the high-performance GPUs consume 250-300W peak power, which is more than twice compared to the CPU counterparts [7]. As technology keeps scaling down, the leakage energy

becomes significant as sub-threshold leakage current increase in CMOS technology [8]. In addition, at very small feature technologies, the imprecise control capability of manufacture process results in process variations (PV), which further cause energy-inefficiency. PV is the divergence of transistor parameters from their nominal values, which induces access speed and minimum supply voltage requirement variations among transistors. In order to ensure the processors run as expected, the maximum clock frequency (F_{max}) and the minimum supply voltage (V_{min}) are limited by the worst case of the whole chip, which leads to substantial performance degradation and more energy consumption. Thus it becomes crucial to address the energy-efficiency challenge in GPUs under technology scaling.

In this thesis, I introduce several architectural approaches to address reliability and energy-efficiency challenge in GPUs, respectively. To summarize, this thesis makes the following contributions:

- I first explore two opportunistic soft-error detection techniques to cost-effectively improve the streaming processors reliability [9]. Observing that the streaming processors are not fully utilized during program execution, I propose to Recycle the streaming processors Idle time for Soft-Error detection (RISE). RISE selectively triggers the redundancy to detect soft errors for a set of warps so that leverages the fully idled streaming processors due to long latency operations and uneven block utilization. It also performs the redundancy for a number of threads in certain warps using the partially idled streaming processors during the branch divergence. Experimental results show that RISE obtains strong capability in improving the SPs soft-error reliability by 43% with only 4% performance loss.

- I further propose to LEverage reSistive Memory (i.e., STT-RAM) to effectively mitigate both the registers Soft-error vulnerability and energy consumption (LESS) in GPUs [10]. STT-RAM consumes extremely low leakage power and is immune to soft error attacks, but it also experiences significantly slower write latency than SRAM. I explore the hybrid STT-RAM and SRAM based register file in LESS, and explore the unique characteristics of GPGPU applications to hide the long write latency to STT-RAM. Experimental results show that LESS is able to mitigate the registers' soft-error vulnerability by 86% and achieve 60% energy savings with only 4% performance loss.

- I then propose a set of techniques to mitigate the susceptibility of GPUs register file to PV [11]. I first develop a novel mechanism that classifies registers into fast and slow categories in the highly-banked register architecture to maximize the frequency improvement. I then leverage the unique features in GPU applications to effectively tolerate the extra access delay to the slow registers. Experimental results show that the proposed techniques are able to significantly optimize the overall GPUs performance by 15% under process variations.

- Moreover, I propose to enable reliable operations in GPU registers under process variations and low supply voltages. I first rigorously model and analyze the impact of process variations and undervolting on the GPU register file's reliability. By leveraging GPU's inherent long register dead-time, I then propose a low-cost and effective faulty-patching design (GR-Guard) that achieve reliable operations from unreliable register file at low supply voltages. Experimental results show that GR-Guard effectively maintains the register reliability with less than 2% performance degradation, while achieving an average of 31% energy savings via aggressive voltage reduction.

- Finally, I propose to design energy-efficient L2 cache for the GPUs. Through vigorous workload characterization and analysis, I observe L2 cache is significantly underutilized due to the lack of inter-CTA (cooperative thread arrays) locality in current GPU applications. Leveraging the unique instruction-level data locality similarity in the SIMT programming model of GPU, I propose a simple design (LoSCache) that leverages this feature to dynamically predict the L2-level data re-reference counts and power-off the "dead" cache lines to reduce L2 energy. Experimental results show LoSCache reduces the L2 cache energy by 64% with only 0.5% performance loss.

Chapter 2

Background: GPU Architecture and Its Programing Model

Figure 2.1 shows an overview of the state-of-the-art GPU Architecture [1]. It consists of a scalable number of in-order streaming multiprocessors (SM) that can access to multiple distributed L2 cache banks in different memory partitions [12, 13, 14, 15], and memory controllers via an on-chip interconnection network. Figure 2.1 also illustrates a zoom-in view of the SM. It contains the instruction cache, fetch and decode logic, instruction buffer, warp scheduler, register file, streaming processors (SPs), special functional units (SFU) , load/store units (LD/ST), L1 data cache, constant and texture caches, and shared memory. In addition to CPU main memory, the GPU device has its own off-chip external memory (e.g., global memory) connected to the on-chip memory controllers.

In this paper, we study the NVIDIA CUDA programming model but some of the basic constructs will hold for most GPU programming models. In CUDA, the GPU is treated as a co-processor that executes highly-parallel kernel functions launched by the CPU. The kernel is composed of a grid of light-weighted threads; a grid is divided into a set of blocks (referred as cooperative thread arrays in CUDA); each block is composed of hundreds of threads. Threads are distributed to the SMs at the granularity of blocks, and threads within a single block communicate via the shared memory and synchronize at a barrier if desired. Per-block resources, such as registers, shared memory, and thread slots in an SM are not released until all the threads in the block finish execution. More than one block can be assigned to a single SM if resources are available.

Threads in the SM execute on the single-program multiple-data model. A number of individual threads (e.g., 32 threads) from the same block are grouped together, called warp. In

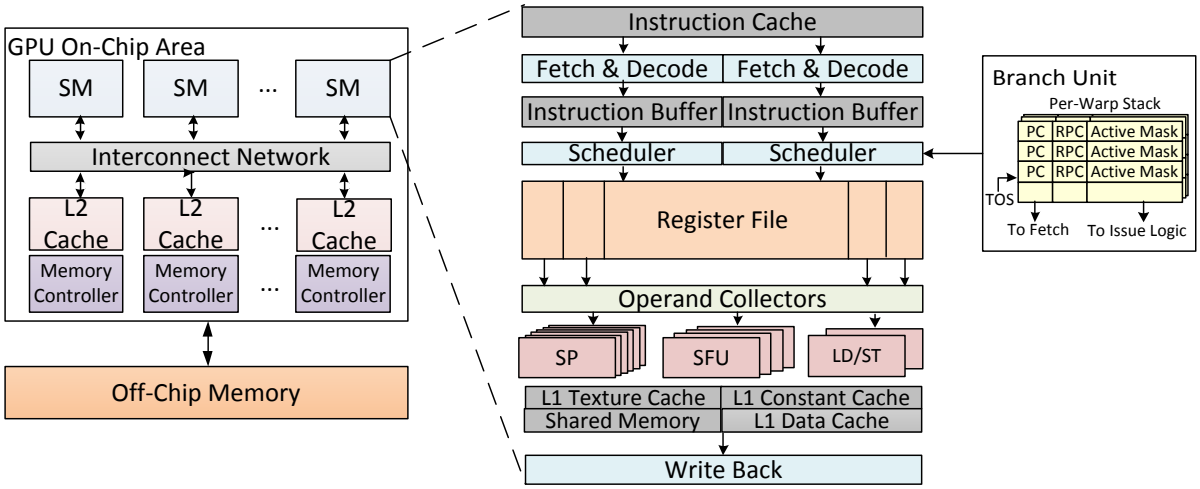

Figure 2.1: An overview of GPU architecture (SM: streaming multiprocessor; SP: streaming processor; SFU: special functional units; LD/ST: load store units).

the pipeline, threads within a warp execute the same instruction but with different data values. Each warp has a dedicated slot in the scheduler which records the warp ID (WID), active mask describing the active threads in the warp, and a single PC. At every cycle, a ready warp is selected by the scheduler to feed the pipeline. The execution of a branch instruction in the warp may cause warp divergence, threads in a diverged warp have to execute in serial fashion which greatly degrades the performance. In the pipeline, a long latency off-chip memory access from one thread would stall all the threads within a warp, and the warp cannot proceed until all the memory transactions complete. The load/store requests issued by different threads can get coalesced into fewer memory requests according to the access pattern.

In NVIDIA PTX standard, an instruction can read up to 4 registers and write 1 register. Therefore, register file (RF) in the SM is heavily-banked (e.g., 16 or 32 banks) instead of multi-ported to provide high bandwidth, and multiple register operands required by one instruction can be read from different banks concurrently [1, 16, 17, 18, 19, 20]. Each RF bank is equipped with dual ports to support 1 read and 1 write per cycle. During the register access, the RF bank ID is obtained based on the warp ID and register ID, and the port attached to that RF bank is activated to serve the access request.

Ideally, the register access for an instruction warp finishes in one cycle [16]. This is not the case when multiple register access requests map to the same bank and cause a bank conflict.

In that case, requests have to be served sequentially, which extends the register access time to multiple cycles and hurts the performance. In order to reduce the possibility of bank conflicts, registers in a warp are distributed across the RF banks. Since multiple source operands for an instruction warp may not be read at the same cycle due to the bank conflicts, operand collectors (shown in Figure 2.1) are applied to buffer the operands. One instruction warp will be allocated one operand collector once issued by the scheduler. When all required operands are ready in its assigned operand collector, the instruction warp proceeds into the execution stage and releases its operand collector resources.

The caches in the GPU are organized hierarchically, composed of SM-private L1 caches and shared L2 cache. The L1 caches are only accessed by threads from the same SM, while the L2 cache is accessed by all threads from the entire kernel. The L1 data cache is typically write-evict and write no-allocate [14, 13], while the L2 cache is write-back with write-allocate [14, 13]. The caches are non-inclusive non-exclusive without hardware coherence [13, 14] to simplify the design.

Chapter 3

RISE: Improving the Streaming Processors Reliability Against Soft Errors

There are several parallel streaming processors (SPs) in each GPU SM. They perform the fundamental computing operations, and play an important role in exploiting the parallel computing. In the GPUs with thousands of parallel threads, SPs execute numerous instructions and expose them to neutron and alpha particle strikes, leading to the high SER. We evaluate the SPs soft-error vulnerability by computing its architecture vulnerability factor (AVF) which estimates the possibility that a transient fault in the structure will produce incorrect computation output, and find that the AVF is 53% on average (detailed description on AVF can be found in Section 3.2). In addition, as the key and representative combinational logic-based structure in the GPUs, SPs occupy a large fraction of the chip area, the SER of SPs becomes the major contributor to the overall SER of the GPU processor. Effectively protecting SPs becomes the essential first step to build the resilient GPU architecture.

To overcome the soft-error reliability challenge, duplication [21] is the well-known classical technique: all instructions can be redundantly executed in the SPs and two copies are compared for error detection. However, the simple duplication will lead to high performance penalty. Based on our experiments on various benchmarks, we found that on average, the full redundancy results in 58% performance degradation. The high computing throughput is the critical feature provided by GPUs. In general, substantially sacrificing the performance to achieve the perfect error coverage is not an efficient solution to the reliability issue in GPUs. A good trade-off between performance and reliability is more acceptable and desirable in future commodity GPU design.

GPU exploits thread-level parallelism (TLP) by grouping a number of threads into a warp

which simultaneously executes the same instruction from each thread, warp are interleaved at cycle-by-cycle basis to hide the latency caused by the data dependence between consecutive instructions from a single warp. However, SPs are unused when all warps stall in the pipeline due to the long latency operations (e.g., off-chip memory access). In addition, threads in the same warp may direct to different paths at a branch, they execute sequentially in the diverged warp and partial SPs in the GPU core become idle [22]. We investigate a large set of GPGPU benchmarks, and find that on average, all SPs in a GPU core are idle during 35% of the total execution time, and the case that the SPs are partially idle appears in 11% of the execution time. The large fraction of the SPs idle time provides great opportunities to trigger the partial redundancy and trade-off little performance degradation for maximal reliability improvement. In this chapter, we propose RISE (Recycling the streaming processors Idle time for Soft-Error detection) which intelligently leverages the under-utilized SPs to perform the redundancy.

The contributions of this chapter are as follows:

- We propose full-RISE that effectively re-uses the fully idled SPs in the GPU core caused by the long-latency memory accesses and the load imbalance among cores for soft-error detection. Full-RISE is composed of the request pending aware Full-RISE (RP-FRISE) and idle streaming multiprocessors (SMs) aware Full-RISE (IS-FRISE). RP-FRISE predicts the warp stall time for its next off-chip memory access and appropriately delays the warp progress via the redundant execution, therefore, successfully recycles the stall time for redundancy without degrading the performance. IS-FRISE estimates the load imbalances among cores, and smartly triggers the redundancy in cores which are predicted to execute fewer threads.

- We propose partial-RISE that redundantly executes a number of threads from a warp by combining their execution with the diverged warp, therefore utilizes the partially idled SPs during the branch divergence for reliability optimization.

- Our experiments show that both full-RISE and partial-RISE are capable of optimizing the SPs soft-error robustness substantially with negligible performance penalty. As the integration of the two techniques, RISE is able to enhance the SPs vulnerability by 43%

with only 4% performance loss. Based on our sensitivity analysis, RISE is also applicable to GPU architecture designs with various performance optimization schemes.

3.1 RISE: Recycling the Streaming Processors Idle Time for Soft-Error Detection

In this section, we propose to intelligently Recycle the stream processors Idle time for Soft-Error detection (RISE). It leverages the SPs idle time to execute the redundant threads and substantially improve the SPs reliability with negligible performance loss. We present the full-RISE and partial-RISE in Section 3.1.1 and 3.1.2, respectively. When implementing the redundant multithreading technique in CPUs, the main and redundant threads share the pipeline resources, no extra hardware resources are required to support the redundant thread. However, every parallel thread in GPU has statically allocated resources including the register files and on-chip shared memory. Those per-thread resources have to be double-sized to successfully launch the main and redundant threads simultaneously into the GPU core, which leads to extremely high resource usage and power consumption [23]. In order to avoid this high overhead, the redundant thread in our study will use the same per-thread resources with the main thread. In other words, the redundant thread follows the main thread immediately, they interleave at the instruction-by-instruction basis and execute at the same speed. In this study, we focus on the single-bit error model which has the first order impact on the failure rate in processors [24]. Note that SPs will operate the computations from main and redundant threads in two consecutive cycles, our technique is sufficient to detect the single-bit errors in SPs. Moreover, it is also able to catch the multiple-bit errors occurring simultaneously in the SPs resulting from either single upset event or multiple, independent upsets. The particle-strikes which could last for more than a cycle in a single bit are not considered in this paper.

3.1.1 Full-RISE

In previous work, the opportunistic transient-fault detection in CPUs has been proposed by Gomma et al. [25]. It keeps the progress difference between the main and redundant threads, and triggers the redundant thread when the main thread stalls for the long-latency operations. Therefore, the redundant thread can efficiently leverage the under-utilized pipeline resources

to perform the redundancy without degrading the performance. However, the technique is not applicable to the SPs soft-error detection in GPUs since both main and redundant threads in GPUs have to keep the same progress (as we described above), and they stall at the same time. A novel technique is desired to intelligently trigger the redundant thread ahead of the pipeline stalls in the SM. In this subsection, we propose Full-RISE which is composed of two methodologies: request pending aware full-RISE and idle SMs aware full-RISE.

3.1.1.1 Request Pending Aware Full-RISE

3.1.1.1.1 The Observation on Resource Contentions among Memory Requests

In CUDA programming model, all threads in a kernel execute the same code. Moreover, warps in the GPU SM interleave at cycle-by-cycle basis and exhibit similar execution progress. When one warp sends out a request for the off-chip memory access, other warps are likely to issue the requests at approximately the same time. The sudden burst of the memory requests will cause the congestion in the interconnect network and the memory controller, leading to the severe resource contentions. Especially, the input buffer in the memory controller will be quickly filled up by the requests. Generally, the out-of-order (OoO) first-ready first-come first-serve scheduling policy is applied in the memory controller. It grants higher priority to the requests which hit an open row in the DRAM, and save the time spent on precharge and activation to open a new row. However, its impact on alleviating the request congestion is ambiguous since a limited number of requests (e.g., only one) can be serviced at a time, and most requests are experiencing the long waiting time.

Figure 3.1 shows a case observed in a CUDA benchmark - BP. A large number of memory requests from the SM are routed to the memory controller in a short period. Due to the serious resource contentions, the off-chip memory access takes up to thousands of cycles. Correspondingly, the SM suffers extremely long pipeline stall, and SPs remain idle (as shown in Figure 3.1(a)). Instead of sending the memory requests at similar time and stalling afterwards for a long period, warps can run at slower pace which prolongs their requests issues (e.g., req2 - reqn in Figure 3.1(b)) and avoids the possible resource contentions, therefore, their requests are serviced without any delay in the input buffer. Due to the dramatically reduced memory operation latency, the total warp execution time still keeps the same or even decreases. This provides the

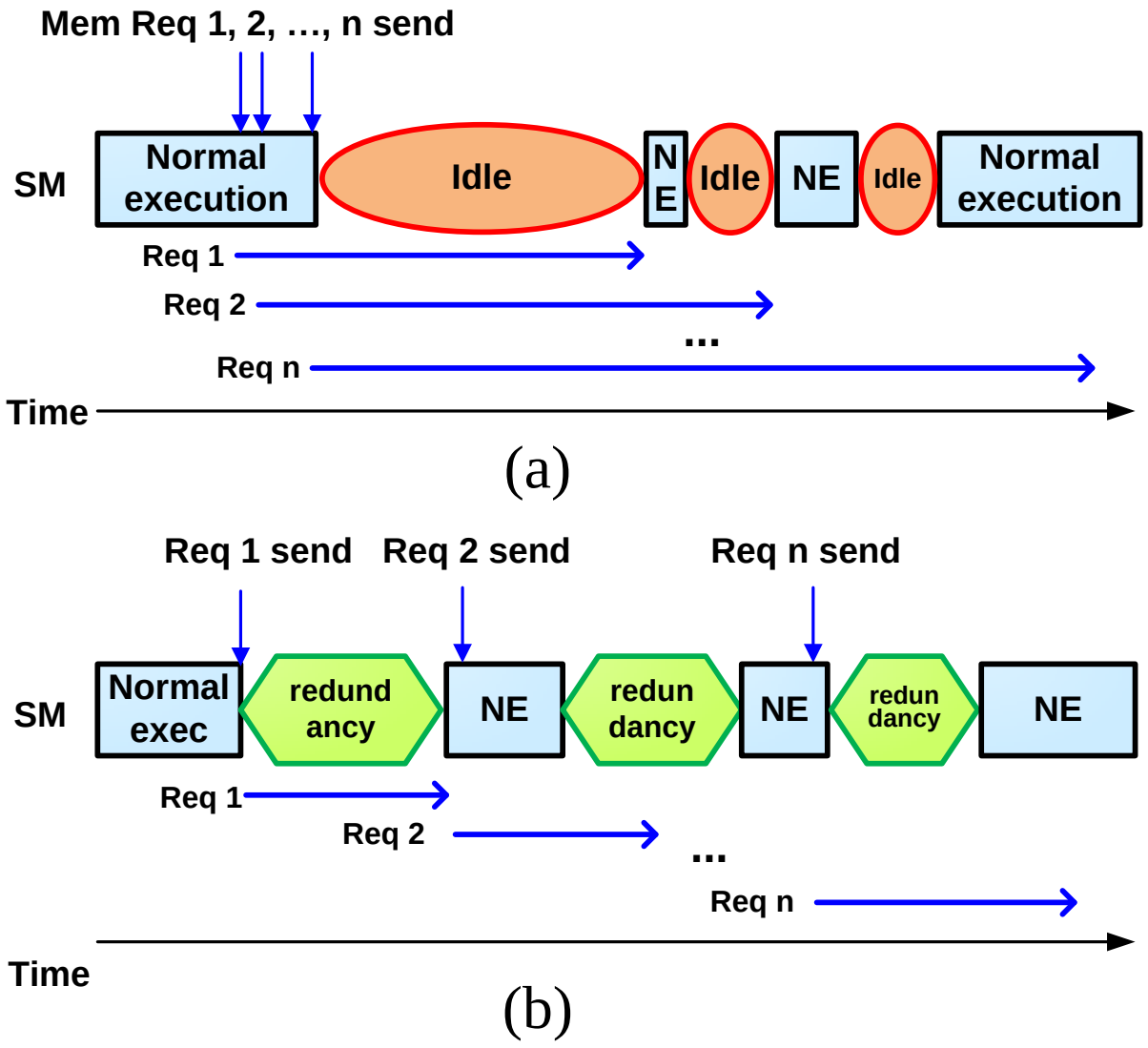

Figure 3.1: SPs idle time (a) caused by resource contentions among memory requests, and (b) recycled for partial redundancy (NE: normal execution).

great opportunity for temporal redundancy, whose negative effect on performance can just be positively used to postpone the warp execution progress and their requests issue time. The SPs idle time caused by the request pending in the memory controller (highlighted with red color in Figure 3.1(a)) is successfully recycled for the redundancy (highlighted with green color in Figure 3.1(b)). Note that the redundant and normal executions are interleaved at cycle level. We mark the redundancy separately from the normal execution in Figure 3.1(b) for the easy understanding of the impact of redundancy. By running a large number of benchmarks, we observe that the request pending contributes to 65% of the time that all SPs in the SM are free. It implies that the SPs soft-error reliability can be substantially improved with no performance penalty.

3.1.1.1.2 The Concept of Request Pending Aware Full-RISE

As discussed above, the partial redundancy (relative to the full redundancy) can be treated as the knob to control the warp progress. It has to be carefully tuned: the over adjustment will result in the excessive redundancy and unnecessarily delay, consequentially, degrades the performance significantly; on the other hand, the insufficient adjustment cannot effectively leverage the SPs idle time for reliability enhancement. Moreover, different warps should spend different amount of time on redundancy in order to separate their memory requests. In one sentence, the warp progress should well adapt to its memory access pattern to achieve the optimal SPs soft-error robustness. Obviously, statically determining the period of performing the partial redundancy fails to meet the goal since workloads have various memory access patterns. In this study, we propose RP-FRISE, as the abbreviation of request pending aware full-RISE, which dynamically tunes the knob (i.e., partial redundancy) per warp and recycles the SPs idle time to maximize the error coverage in SPs.

Since the major resource contentions occur in the memory controller, the request waiting time in the input buffer is a good indicator to the necessity of slowing down the warp. A long waiting time implies a serious resource contention, the warp which issues the request should have been delayed. While a short waiting time means that the request is issued appropriately, postponing the warp progress may degrade the performance. In the ideal case, a memory request

gets serviced once it arrives at the memory controller: the period that allows a warp to perform the redundant execution should be equal to its request waiting time. In RP-FRISE, we use the previous request pending time to predict the delay in the following memory transaction and tune the partial redundancy in the warp.

Note that the warp progress has already been postponed somehow when finishing the previous memory access, further slowing it down as the same amount of the previous request waiting time may serious prolong the warp computation. An example is shown in Figure 3.2 to illustrate the challenge: *warp0* and *warp1* exhibit different execution progress after the first memory access, and their following memory requests have less interference. Using the previous request waiting time leads to excessive redundancy and degrades the performance. On the other hand, although warps run at different rate after the long latency memory access, the interference is still severe. Figure 3.3 demonstrates the case. When the second memory request is issued from *warp0*, it interferes with the unfinished memory access from *warp1* and the waiting time increases. The extended memory access in *warp0* further affects the *warp1*, and setting the redundancy time as the preceding request waiting time for *warp1* is appropriate. Note that *warp1* may not delay as long as the scheduled redundancy time even performing the full redundancy for the computation between its two memory accesses (highlighted by the pink color in Figure 3.3). In that case, it will be stalled after finishing the redundancy due to the interference with *warp0*. Therefore, the required redundancy time still accurately predicts the necessary delay in *warp1*. As can be seen from Figures 3.2 and 3.3, generally, the excessive redundancy occurs when the kernel is more computation-intensive and the time spent on the normal computation alleviates the memory contentions among threads. While the short normal computation in the memory-intensive kernel cannot effectively separate the memory requests, a longer redundant execution is desired.

In our study, we sample the memory access latency periodically when running a kernel, and use it to adjust the amount of cycles that the warp executes the redundant threads. Eq. 3.1.1.1.2 describes the analytical model to dynamically determine the redundancy cycles (represented by RC) in the warp based on its previous request pending time (represented by T_P) and the latest

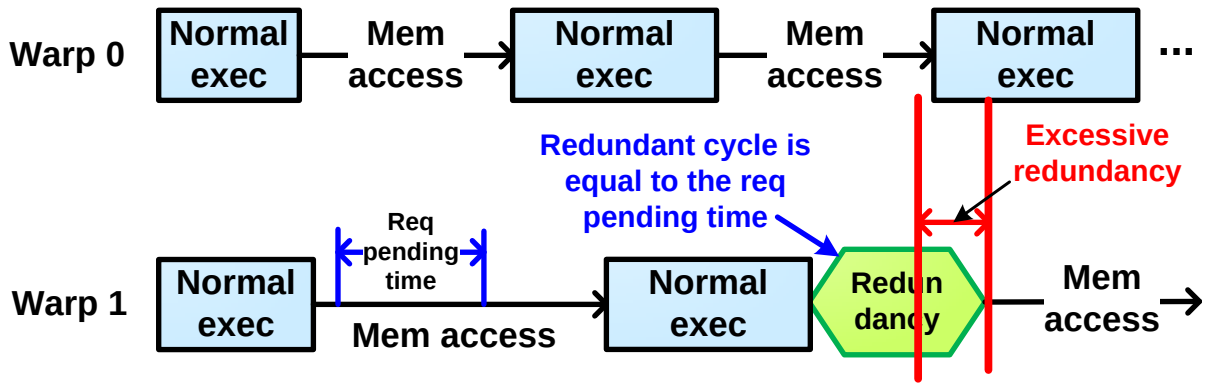

Figure 3.2: An example of the excessive redundancy.

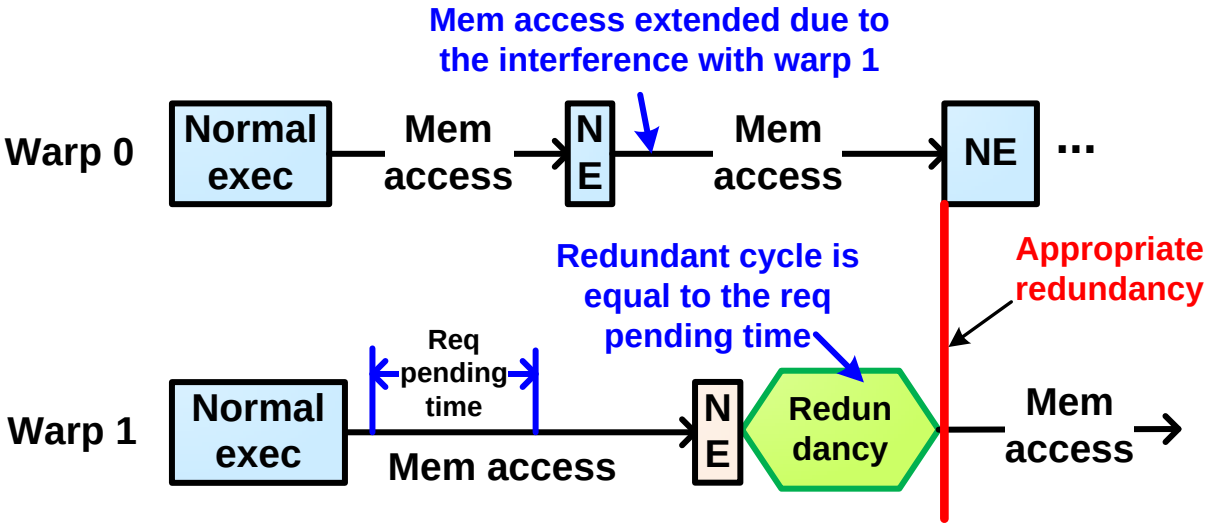

Figure 3.3: An example of the appropriate redundancy.

sampled memory access latency (represented by T_{acc_lat}) by

$$RC = \begin{cases} 0 \quad (\text{if} \quad T_P \leq T_{thr\_pend}) \\ \left[\frac{T_{acc\_lat}}{T_{ref\_lat}} \times T_P\right] \quad (\text{if} \quad T_{acc\_lat} < T_{ref\_lat} \quad \text{and} \quad T_P > T_{thr\_pend}) \\ T_P \quad (\text{if} \quad T_{acc\_lat} \geq T_{ref\_lat} \quad \text{and} \quad T_P > T_{thr\_pend}) \end{cases}, \tag{3.1}$$

where T_{thr_pend} is the threshold of the request pending time. It is possible that the previous memory access is over delayed by the redundancy, the improper delay (implied in the prior request pending time) cannot further propagate to the following execution. T_{thr_pend} plays an important role to filter it out. Only when T_P is longer than T_{thr_pend}, the warp redundancy would be enabled, otherwise, it is disabled to maintain the performance. T_{ref_lat} is the referred memory access latency describing the memory access time with moderate resource contentions. When T_{acc_lat} is higher than T_{ref_lat}, it implies that the kernel currently exhibits the memory-intensive characteristics and the aggressive redundancy (i.e., directly setting the redundancy cycles as the request pending time) should be applied. To the contrary, a lower T_{acc_lat} means that the kernel involves heavier computation, the redundancy period should be scaled down according to the ratio of T_{acc_lat} to T_{ref_lat}.

Recall that threads in a warp execute the same instruction, triggering the redundancy at the warp level becomes the first choice in RP-FRISE. However, all threads in the block have to synchronize at barriers if exist, large progress difference among warps belonging to the same block will extend the faster warps' waiting time at the barrier and hurt performance. Additionally, GPGPU programmers are encouraged to make consecutive threads access consecutive memory locations, warps in a block tend to show the strong spatial locality [1, 26]. The memory requests issued by those warps are likely to be directed to the same row in the DRAM, called row locality [22]. When they are sent out simultaneously, the pending time in the input buffer of the memory controller tends to be similar under the out-of-order memory scheduling policy. On the other hand, it will even take longer time to complete the memory transactions if issued separately, because the row locality among warps is broken and the row switches more frequently. In order to maintain the performance, RP-FRISE performs the block level redundancy. Since the redundant time is computed on the basis of memory request, a single warp may have more than

one option on setting the redundancy cycles, and there are numerous choices when extending to a block. In RP-FRISE, the redundancy time will be incorporated into the response packet from the memory to the SM. To a block, the first arrived packet after it finishes the previous assigned redundant execution sets the new redundancy cycles, which should be applied to all its warps.

3.1.1.2 Idle SMs Aware Full-RISE

While a kernel is launched into the GPU, its blocks may not be even distributed across the SMs. A number of SMs are free when approaching the end of the kernel execution, and they have to wait for other busy SMs to finish their tasks. In other words, the SPs become idle when no more blocks can be assigned to the SM. We found that this case contributes to 35% of the time that all SPs are free in the SM. Those free SPs can be leveraged for redundant execution. One straightforward method is to redundantly execute the blocks which are currently running in other SMs. It will cause the challenges for memory synchronization between the original and redundant blocks and introduce more memory transactions from the redundant blocks.

Instead of implementing the redundancy when the SM is free, we propose to do it at the beginning of the kernel execution so that the SMs execution time is aligned and the SPs idle time is effectively recycled for redundancy. This technique is named as IS-FRISE, as the abbreviation of idle SMs aware Full-RISE. Based on the information obtained during the kernel launch process (e.g., the total number of blocks, the maximum concurrent blocks a single SM supports when running the specific kernel), a simple calculation is done to roughly estimate the total number of blocks assigned to each SM through the entire kernel execution ignoring the possible memory access delay. We conservatively assume that there is only one block difference among SMs, and divide the block quantity by the number of SMs. If there is a remainder, we expect that some SMs may be free at the end of the kernel execution, and the remainder determines the quantity of SMs (which are randomly selected among all the SMs) running an additional block. Once the kernel is launched, the full redundancy is applied to one of the currently executed blocks in SMs which are predicted to run fewer blocks, thus, the total loads including the redundancy are balanced across SMs and performance remains the same. Note that the load imbalance among SMs can be larger than one block since faster SMs will take more blocks, but predicting

a large load difference is likely to cause the overestimate of the SMs idle time which leads to the aggressive redundancy and hurt the performance.

When integrating the two full-RISE techniques simultaneously, some blocks may perform the redundancy twice which is a waste of resources. Considering that IS-FRISE only takes in effect on a small set of blocks, the redundant execution scheduled by RP-FRISE on those blocks will be ignored. Although there is an overlapped effect between the two techniques, their positive interaction minimizes the potential of excessive redundancy in RP-FRISE. Recall that RP-FRISE depends on the previous request pending time to determine the redundancy cycles for the following execution, it loses the opportunities to perform redundant execution before the blocks issue their first memory requests. The memory contentions among the first memory transactions tend to be severe. Using the first request's pending time for redundancy cycles calculation would over delay the block progress, and this negative effect is likely to propagate towards the end of the kernel execution. IS-FRISE triggers the full redundancy on certain blocks at the very beginning of the kernel execution, it differentiates the block progress across SMs and mitigates the resource contentions even before the first memory requests, hence, effectively reduces the possibility of the unnecessary redundancy.

3.1.1.3 The Implementation of Full-RISE

Figure 3.4 shows the implementation of the request pending aware full-RISE in the GPU architecture. In the memory controller, a timer is attached to each input buffer entry, and an arithmetic logic unit (ALU) is added to calculate the redundancy cycles (in our study, we assume that only one request is serviced at a time, and one ALU is sufficient to perform the calculation). When a memory request is written into the input buffer, the timer is set as zero and automatically increments every cycle, it records the request pending cycles which will be sent to the ALU when the request is issued out for DRAM access. The sampled memory access latency, threshold pending time and referred access latency are used as the inputs to the ALU as well. Its output is combined with the response packet and sent back to the SM. Considering that ALU operation has to be done per memory request, and the major computation in it is the division (as shown in Eq. 3.1.1.1.2.) which lasts for tens of cycles, we set the referred access latency as 2 to power of n and translate the division into logical shift. It will operate

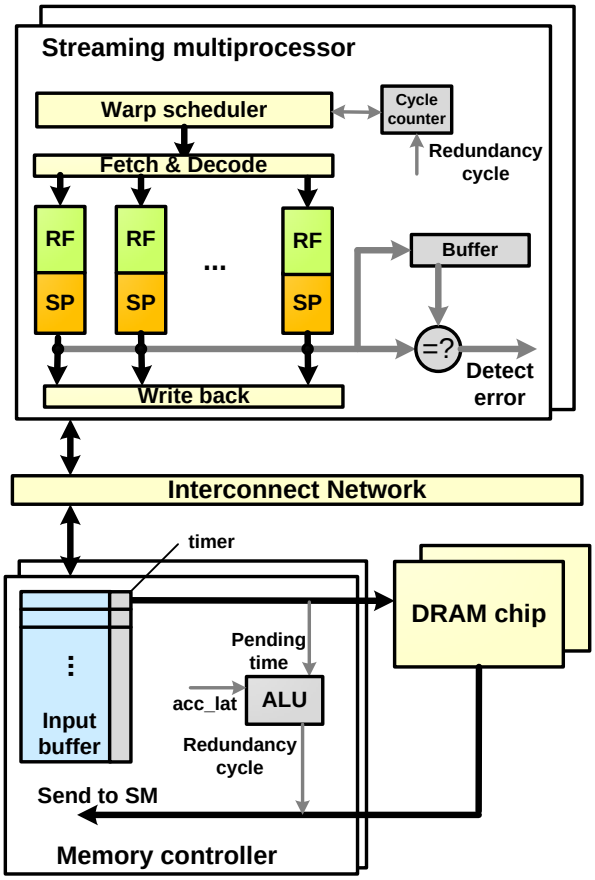

Figure 3.4: The implementation of request pending aware full-RISE.

based on the product of the average access latency and the pending time. We performed the detailed sensitivity analysis on the referred access latency, and found that RP-FRISE achieves optimal trade-off between reliability and performance when setting it as 2^7=128 cycles. Note that the redundancy time computation occurs in parallel with the data access in DRAM, it does not introduce any extra delay to the critical path in the memory controller. As Figure 3.4 shows, each block in the SM is allocated a cycle counter which keeps the redundancy cycles. The amount of counters per SM is determined by the maximum number of concurrent blocks an SM supports (e.g., 8 in our default configuration). When a response packet arrives at the SM, we can find out its corresponding cycle counter based on the warp it returns to. If the counter is larger than zero, it implies that the block is already under the redundancy mode, the new redundancy time (in cycles) will be ignored. Otherwise, it is multiplied by the number of warps in the block, and the result will be written into the counter, which implies the desired total amount of redundancy cycles applied to all warps in the block. In this study, instead of

controlling each warp in a block to spend the same amount of time in redundancy mode, we apply a relaxed mechanism to manage the total redundancy cycles at the block level. When a ready warp is selected to feed the pipeline, the counter is accessed, a larger-than-zero value suggests a redundant execution. The warp will remain in the warp scheduler and be granted a high issue priority to ensure that the same redundant warp is executed in the following cycle. Meanwhile, the counter decreases by one. During the write back stage, the SPs outputs of the original warp are written into the destination registers and an attached buffer; while the redundant warp's outputs will skip the write operation, and directly compare with the data just written into the buffer for the error detection. A mismatch will raise the recovery signal. The warp PC will be fetched again to start additional redundant execution, three copies' comparison will correct the error and resume the warp computation.

The idle SMs aware full-RISE requires modulo operation to determine the number of SMs running fewer blocks. A simple AND logic operation can compute out the remainder. It performs simultaneously with the kernel launch process, and no extra delay is introduced to the normal kernel execution. When combining the two full-RISE mechanisms, the block with full redundancy selected by the IS-FRISE will set its counter as one and disable the value decrease function until it finishes, ensuring the full redundant execution for its warps.

As can be seen in Figure 3.4, the major hardware added into the memory controller includes the unit performing simple integer arithmetic and logic operations, and a number of 10-bit timers (we set the maximum request pending time as 1024 cycles), it causes around 3% area overhead to the memory controller. In the SM, the added hardware contains 8 cycle counters, thirty-two 32-bit buffers for warp outputs (the SM pipeline width is 32 in the default configuration, each SP output 32-bit value), and some combinational logics, totally resulting in 1% area overhead to the SM.

3.1.2 Partial-RISE

3.1.2.1 The Concept of Partial-RISE

When the warp diverges at the branch instruction, several SPs in the SM are idle. In the workload encountering frequent branch divergences, SPs are partially free in majority of the time. For instance, the case occurs during 52% of the total kernel execution time in a CUDA

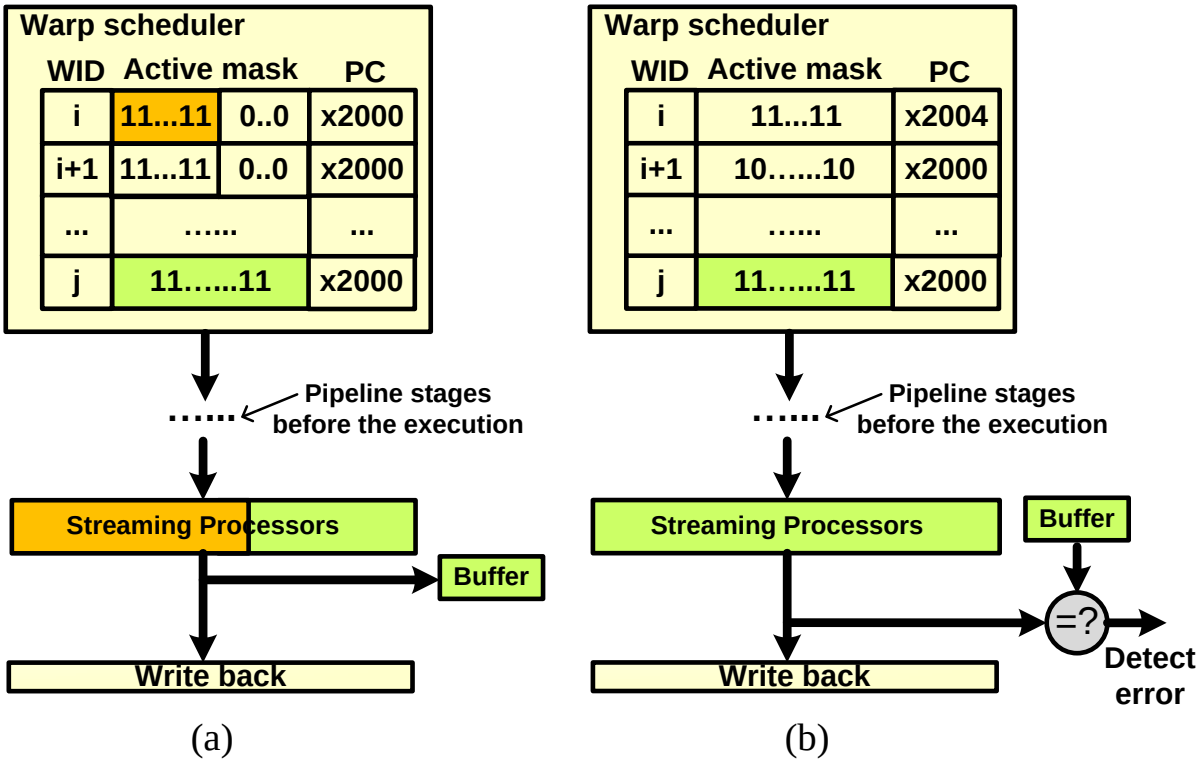

Figure 3.5: An example of partial-RISE.

benchmark - HS. Unfortunately, full-RISE fails to leverage such large portion of SPs idle time for reliability optimization due to its nature of performing the redundancy via using all SPs. In this subsection, we propose partial-RISE which intelligently combines the redundant threads into the diverged warp to utilize the partially idled SPs, thus, improves the SPs error coverage and maintains the performance. As described in Section 3.1.1, GPU has the unique microarchitecture characteristic (e.g., warps interleave at cycle level, and all threads execute the same code), and typically, there are numerous warps in the SM warp scheduler. When a ready warp is issued into the pipeline, it is highly possible that another ready warp sharing the same PC is sitting in the scheduler. Similarly, the diverged ready warp can usually find another ready warp (not necessarily diverges), and both are going to perform the same operation in SPs. A number of threads from a warp with the same PC can join the execution of the diverged warp. Therefore, the warp is partially protected as the idle SPs in the diverged warp are effectively utilized to execute the redundant threads for it. Moreover, the warp will be issued in the following cycle so that both main and redundant threads output can be compared immediately for the error detection. We name this technique as partial-RISE. Figure 3.5 demonstrates an example of it.

In Figure 3.5(a), at cycle m, the warp i and j have the same PC, and the active mask shows that warp i diverges. When it is issued into the pipeline, threads from warp j will take the free slots in warp i based on its active mask. When the outputs of warp j are available, they will be sent to the buffer instead of writing to the registers. During the cycle m+1(shown in Figure 3.5(b)), warp j is issued, it outputs are compared with the data saved in the buffer in previous cycle.

Finding the warp with the same PC is critical to partial-RISE. We investigate various benchmarks, and Figure 3.6 plots the ratio of the number of cycles that at least one ready warp has the identical PC with the currently issued diverged warp to the total warp divergence cycles (shown as the red bars). The round-robin policy is applied for the warp scheduling. The benchmarks with quite few branch divergences are not shown in the figure since the partial-RISE is rarely triggered in that case. As it demonstrates, in more than half of the time, the diverged warp has the opportunity to combine with another warp by searching across the warp scheduler. However, the two warps are likely to use the SP and registers belonging to the same lane and encounter the lane conflict. It becomes the major obstacle to partial-RISE. As shown in Figure 3.5(a), although warp i+1 has the identical PC with warp i, partial-RISE cannot be used as both them have the same active mask. Figure 3.6 also shows the possibility that a diverged warp can be combined with another warp without lane conflict (shown in the yellow bars). As can be seen, it decreases significantly down to 16%, and even becomes zero in some benchmarks (e.g., BP, SLA, and ST3D). Since threads in a warp are independent but their operations are the same, there is no requirement to bind a thread to a certain lane. We propose to randomly shuffle the threads while sending them to the SMs, it will be performed in parallel with the kernel launch process and no extra delay to the entire kernel execution. By doing that, the possibility of lane conflicts between two warps decreases and partial-RISE can be triggered more frequently. The black bar in Figure 3.6 shows that re-arranging threads successfully brings the possibility of finding a ready warp with the same PC back to 48%.

Recently, several mechanisms (e.g., the dynamic warp formation [27], thread block compaction [28], the large warp microarchitecture (LWM) [22]) have been proposed to improve the efficiency of branch handling. Take the LWM as an example, it implements larger warp so that

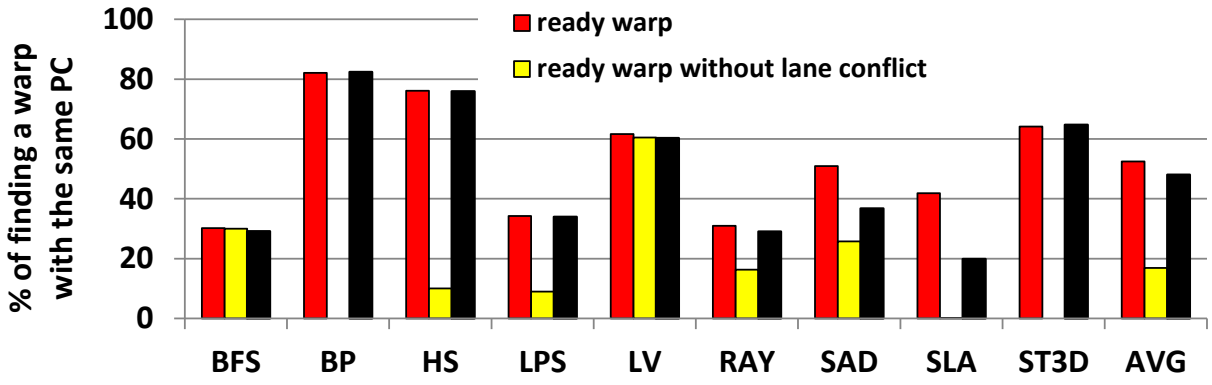

Figure 3.6: Possibility that a diverged warp finding a warp with identical PC.

the sub-warps can be formed from the active threads in a large warp, and when they are executed in the pipeline, the SPs idle time reduces under branch divergences. One possible concern is that partial-RISE has trivial benefit for reliability enhancement when LWM is applied in the GPU. This is not the case. A large warp contains much smaller number of threads (e.g., 256) compared to the entire warp scheduler. It is unable to find active threads out of the warp, and it does not provide any mechanism to avoid the lane conflicts. LWM does not always fully utilize the idled SPs during branch divergences. While partial-RISE searches warps across the entire scheduler, and is equipped with the thread shuffling technique, it can efficiently use those idled SPs in LWM for redundant execution. We observe that partial-RISE takes in effect in 20% of the time when LWM fails to fully utilize the SPs.

In some benchmarks (e.g., NN, NW), the block contains few threads (e.g., 16) so that it has only one warp and all threads in that warp cannot fill up the SIMT pipeline width. Since several lanes keep idle through the entire kernel execution, partial-RISE will perform the intra-warp duplication which leverages those idle lanes to provide the spatial redundancy for some threads contained in the warp.

3.1.2.2 The Implementation of Partial-RISE

In the SM pipeline, when the warp enters to the final pipeline stage (i.e., write back stage), its PC and active mask are updated, and its status turns to be ready for issue in the following cycle. To implement the partial-RISE technique, a comparator is attached to each warp entry in the warp scheduler. While updating the warp status, its PC (active mask) will be compared with other ready warp PCs (active masks) in the scheduler to seek a warp for joined execution.

The comparison is executed along with the write back stage, and no impact to the critical path delay. If succeeds, the scheduler will be notified to issue the diverged warp and threads from its matched warp simultaneously in the next cycle, followed by the normal issue of the matched warp. Partial-RISE will re-use the hardware (e.g., buffer, comparator) in full-RISE to perform the error detection. In total, partial-RISE leads to additional 1% area overhead to the SM equipped with full-RISE.

3.1.3 RISE: Putting It All Together

Since full-RISE and partial-RISE target to recycle the SPs idle time caused by two different cases for reliability improvement, they can be integrated into RISE. While implementing RISE, the warp under the redundancy mode due to the full-RISE will not be considered in partial-RISE. Although full-RISE differentiates the block execution progress, it does not degrade the efficiency of partial-RISE in finding appropriate warp for redundancy. We find that the partial-RISE trigger time even increases by 2% when combined with full-RISE. It is because that the row locality leads to the similar redundancy cycles, and consequently, similar progress among blocks in the same SM under full-RISE, and the block progress difference generally happens at the SM level.

3.2 Experimental Setup

We use architecture vulnerability factor (AVF) [29] to evaluate the error coverage of our proposed RISE. A hardware structure's AVF refers to the probability that a transient fault in that hardware structure will result in incorrect program results. Therefore, the AVF, which can be used as a metric to estimate how vulnerable the hardware is to soft errors during program execution, is determined by the processor state bits required for architecturally correct execution (ACE). The structure's AVF in a given cycle is the percentage of ACE bits that the structure holds, and its overall AVF during program execution is the average AVF at any point in time. We apply the methodology proposed by Mukherjee et al. [29] to identify the ACE bits and their residency time in the structure and compute the AVF of GPU microarchitecture structures. We build our vulnerability estimation framework based on the cycle-accurate, open-source, and publicly available simulator GPGPU-Sim [19], and obtain the GPU reliability and performance statistics. Our baseline GPU configuration is set as follows: there are 28 SMs

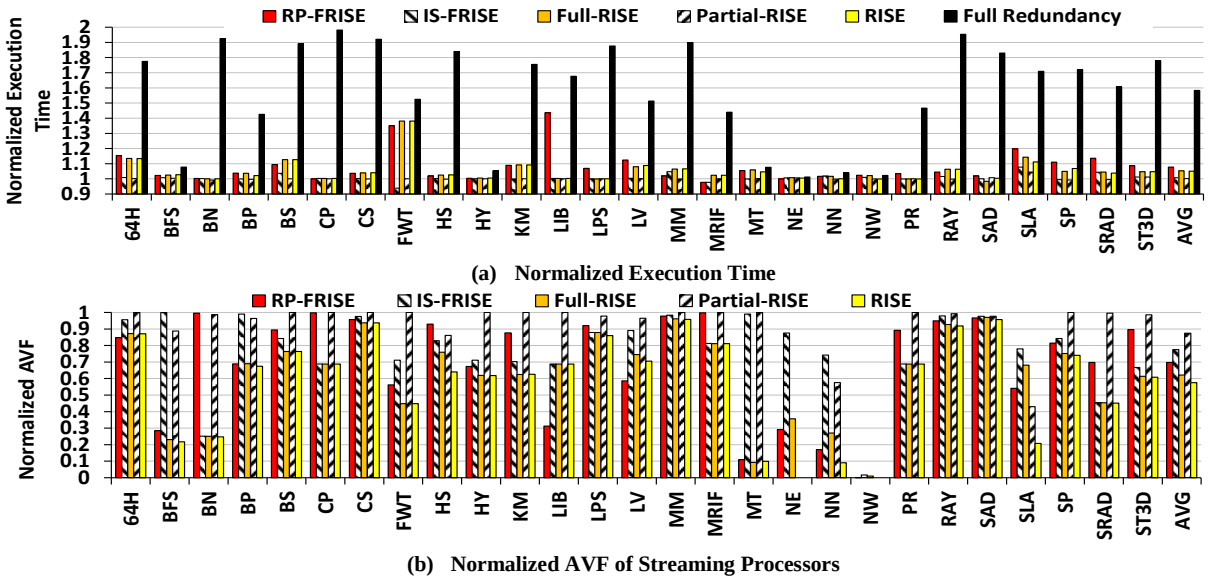

Figure 3.7: The normalized (a) execution time and (b) SPs' AVF under IS-FRISE, RP-FRISE, Full-RISE, Partial-RISE, RISE, and Full Redundancy techniques.

in the GPU, SM pipeline width is 32, warp size is 32, each SM supports 1024 threads and 8 blocks at most, each SM contains 16K 32-bit registers, 16KB shared memory, 8KB constant cache, and 64KB texture cache, the warp scheduler applies the round robin scheduling policy, the immediate post-dominator reconvergence [30] is used to handle the branch divergences; the GPU includes 8 DRAM controllers, each controller has a 32-entry input buffer, and applies out-of-order first-ready first-come first-serve scheduling policy; the interconnect topologies is Mesh, and the dimension order routing algorithm is used in the interconnect. We collect a large set of available GPGPU workloads from Nvidia CUDA SDK [31], Rodinia Benchmark [32], Parboil Benchmark [33] and some third party applications. The workloads show significant diversity according to their kernel characteristics, divergence characteristics, memory access patterns, and so on.

3.3 Evaluation

3.3.1 Effectiveness of RISE

To better analyze the technique effectiveness, we classify the benchmarks into four categories based on their workload characteristics. The first category includes memory-intensive benchmarks such as 64H, BFS, BP, HY, LIB, LV, MT, NE, NN, NW, PR, and SLA. The second

category contains benchmarks which cause load imbalance across SMs in our baseline GPU configuration, they are BN, CP, FWT, HY, KM, LV, MRIF, NW, PR, SLA, SP, SRAD, and ST3D. The third category includes benchmarks usually utilizing partial SPs (caused by the frequent branch divergences or the partially full warps) such as BFS, BP, HS, LPS, LV, NN, NE, NW, SAD, SLA, and ST3D. The last category includes the computation-intensive benchmarks such as BS, CS, MM, and RAY. Note that the categories described above are not exclusive (i.e., one benchmark can be classified into different categories).

Figure 3.7 shows the (a) execution time and (b) the AVF of streaming processors when running various benchmarks under the impact of IS-FRISE, RP-FRISE, Full-RISE, Partial-RISE, and RISE, respectively. The execution time of full redundancy is demonstrated in Figure 3.7(a) as well for comparison. As can be seen, on average, it results in 58% performance degradation. Since the full redundancy achieves 100% error coverage (i.e., AVF is zero), its results are not shown in Figure 3.7(b). We present the averaged results across SMs, and the results are normalized to the baseline case without any optimization. As Figure 3.7 shows, RP-FRISE exhibits strong capability in improving the SPs soft-error vulnerability with little performance loss when executing the memory-intensive benchmarks (classified as the first category). Take the MT as an example, the SPs' AVF decreases by 90% with only 4% performance penalty. It implies that the redundancy cycles are properly set by RP-FRISE and the SPs idle time in the baseline cases is effectively recycled for redundant execution. One may notice that the AVF reduction under RP-FRISE is less obvious in PR, although the warps spend 17% of the execution time in waiting for the memory transactions. It is because their memory requests have already well separated during the execution, the memory access latency is generally short, and the pipeline only stalls couple of cycles for the memory transaction. Postponing the warp progress would easily hurt the performance substantially, therefore, RP-FRISE is seldom triggered. While improving the SPs' AVF, RP-FRISE degrades performance in some benchmarks such as FWT, LIB, and SLA. Because RP-FRISE uses the last request pending time to predict the next request waiting time and determine the redundancy cycles correspondingly, its prediction accuracy is affected when the next memory access pattern differs greatly from the last one. As a result, the excessive redundancy is applied which hurts the performance. On average across the benchmarks in the

first category, RP-FRISE enhances the SPs soft-error robustness by 57% with 8% performance loss.

The impact of IS-FRISE on improving the SPs' error coverage is impressive in benchmarks belonging to the second category. For example, the SPs free time caused by the imbalanced block distribution is completely recycled for redundancy in BN and the AVF decreases 75% with 0.1% performance loss. Recall that IS-FRISE conservatively assumes one block difference among SMs to maintain the kernel execution time, while the difference is larger in some benchmarks (e.g., LV). In the GPU, a block is assigned once there is an empty slot in certain SM. It is possible that an SM commits multiple blocks at similar time and the remaining unexecuted blocks are all allocated to it. Although other SMs finish the block execution in the very near future, they have to remain idle till the kernel completes. In that case, IS-FRISE does not fully leverage the idle SPs to perform redundancy. Monitoring and predicting the block progress in each SM and dynamically controlling the number of redundant blocks may lead to better reliability improvement, but it induces complicated hardware design which should be avoided. On average across benchmarks in the second category, IS-FRISE reduces SPs' AVF 37% with no performance loss. Note that the load imbalance of a benchmark is related to the GPU configuration (e.g., number of SMs), it may not be an issue when configuring the machine differently, but meanwhile, another set of benchmarks may encounter this problem. Therefore, IS-FRISE is applicable to various GPU architecture designs. As the combination of RP-FRISE and IS-FRISE, Full-RISE keeps their positive effects on optimizing the soft-error robustness substantially, and more important, their interaction effectively mitigates the performance penalty caused by the RP-FRISE. As Figure 3.7 shows, Full-RISE improves SPs AVF 39% with 4% performance loss on average across all investigated benchmarks.

The major benefit of Partial-RISE is observed in benchmarks classified into the third category. As shown in Figure 3.7, on average across benchmarks in the third category, it reduces SPs AVF by 32% without any performance penalty. Finally, when putting all together, RISE integrates the benefit of all the proposed techniques by achieving 43% SPs soft-error reliability enhancement and minimizes (only 4%) the performance loss. In the computation-intensive benchmarks, the effect of RISE is less impressive due to the limited SPs idle time. Note that,

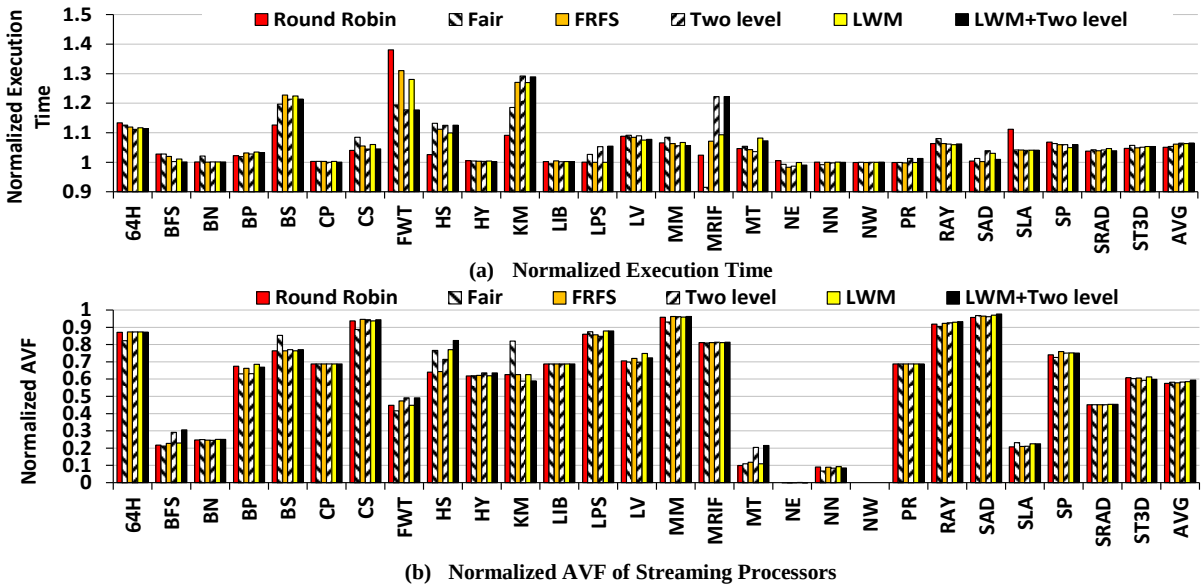

Figure 3.8: The normalized (a) execution time and (b) SPs' AVF under various scheduling policies and performance optimizations.

RISE optimizes the SPs vulnerability via redundancy, the SPs AVF does not migrate to any other microarchitecture structures.

3.3.2 Sensitivity Analysis

Various techniques have been proposed on warp scheduling and warp formation to improve the GPU throughput, such as Fair which issues the instruction for the warp with minimum number of instructions executed [34], First-Ready First-Served (FRFS), large warp microarchitecture (LWM), and two-level round-robin warp scheduling that effectively hides long memory access latency and improves the SPs utilization [22]. Figure 3.8 shows (a) the execution time and (b) SPs' AVF under RISE when those optimization schemes are enabled. Results are normalized to the baseline case with the corresponding optimizations, respectively. The results obtained when using the default scheduling policy (i.e., Round Robin) is also shown in the figure for comparisons. As it shows, the effectiveness of RISE is not affected when running with different schemes: on average, the SPs AVF reduction keeps around 42% with 6% performance loss. Take the scheme of LWM+two-level as an example, it reduces the SPs idle time to some degree to shrink the kernel execution time, one would expect that it largely diminishes the opportunities to trigger RISE. However, this only occurs in a limited number of benchmarks (i.e., BFS, HS, MT) which shrinks the reliability optimization by around 12%. As we described

in Section 3.1.2, LWM only finds active threads in the warp scope and cannot avoid the lane conflicts, it leaves sufficient room for Partial-RISE in RISE to further use the partially idled SPs for redundant execution. Moreover, the two-level round robin scheduling cannot totally avoid the memory contentions and the load imbalance across SMs, therefore, the Full-RISE in RISE can be frequently triggered to recycle the SPs free time for reliability enhancement.

Chapter 4

LESS: Soft-Error Reliability and Power Co-Optimization for Register File using Resistive Memory

Recently, resistive memory, such as spin-transfer torque RAM (STT-RAM), emerges as the promising candidate for future universal memory. It has been proposed in the GPU microarchitecture design [35, 36]. STT-RAM offers several benefits such as extremely low leakage power, high density, non-volatility, and competitive read time as compared to SRAM. Besides these, its advantage of the immunity to soft error attacks [37, 38, 39] provides the great opportunity to build robust and low-power storage-cell based structures, such as the register file, in GPUs.

In this chapter, we propose to LEverage reSistive Memory to effectively mitigate both the registers Soft-error vulnerability and energy consumption (LESS). Although it exhibits the power and reliability advantages, STT-RAM experiences significantly slower write latency than SRAM [40, 41, 42]. Since register file is the crucial structure on exploring the thread-level parallelism, its access time affects the GPU performance [16]. In addition, the long-latency write on RF will occupy the result bus resources for multiple cycles, which can cause pipeline stalls [19]. Therefore, building a pure STT-RAM based register file will cause significant performance loss. We explore the hybrid STT-RAM and SRAM based register file in LESS, and explore the unique characteristics of GPGPU applications to hide the long write latency to STT-RAM and obtain the win-win gains: achieving the near-full soft-error protection for the register file, and meanwhile substantially reducing the power consumption with negligible performance loss.

The contributions of this work are as follows:

- We observe that GPGPUs workloads usually contain a small amount of long-lifetime

register values, and surprisingly, they account for around 90% of the register soft error vulnerability. We then build the STT-RAM and SRAM hybrid register file and propose lifetime-aware LESS to distribute those long lived values to STT-RAM based registers. It mitigates the register SER tremendously with limited write times to STT-RAM, which is able to alleviate the performance penalty and power overhead caused by the STT-RAM writes.

- Not all register values are 32-bit long in GPGPUs applications. We find that more than half of the STT-RAM writes only use the lower 16-bits, named as narrow-width values. We then propose narrow-width-aware LESS to combine two narrow-width STT-RAM writes to share the result bus in order to reduce the performance overhead caused by STT-RAM writes.

- Our experimental results exhibit that the integrated LESS technique successfully builds a soft-error robust (86% vulnerability mitigation) and low-power (60% energy savings) register file in GPUs with only 4% performance loss. To our best knowledge, this is the first work to leverage the advantage of STT-RAM on particle strikes in building the soft-error robust and low-power GPUs.

4.1 Background: Resistive Memory and Its Immunity to Soft Errors

STT-RAM uses the Magnetic Tunnel Junction (MTJ) as the data storage device, instead of latching circuitry [40, 41, 42]. The MTJ is composed of two ferromagnetic layers that are separated by one tunnel barrier layer. One of the ferromagnetic layers (the reference layer) has fixed magnetic direction, while the other one (the free layer) can change its magnetic direction by applying a large current. When the two ferromagnetic layers have the same direction, the MTJ resistance is low, indicating the "0" state; the different direction of the two layers means a high resistance in the MTJ and indicates the "1" state.

The STT-RAM cell is composed of one NMOS transistor and one MTJ. The NMOS is used as the access device, and connected in series with the MTJ. When writing "1" ("0") into the STT-RAM, a large positive (negative) voltage is applied between the source line and the bit line,

leading to the longer write delay and higher write energy compared to the write operation in the SRAM cell. A read operation in the STT-RAM only requires a small sense current injecting to the cell, and the read delay is competitive to SRAM. Besides its fast read, STT-RAM also exhibits the near-zero leakage due to the use of non-volatile storage cell.

When a high-energy neutron or alpha particle strikes the SRAM-based storage cell, the device state flips and a soft error occurs once the amount of the charges caused by the particle is higher than a certain threshold. It has been observed that a particle is not able to generate sufficient charges and switch the state of a MTJ under various altitudes [38]. In other words, different from SRAM which is susceptible to the particle strikes, STT-RAM is immune to the radiation induced soft errors as a data storage device. There have been many studies leveraging STT-RAM to build the soft-error resilient microarchitecture [38, 37]. So STT-RAM is generally recognized as soft-error free [38, 37, 39]. In this work we assume the data stored in the STT-RAM is free from radiation induced soft errors.

4.2 LESS: Leveraging STT-RAM to Build Soft-Error Robust and Low-Power Register File

In this section, we propose LESS to take advantage of STT-RAM's immunity to soft errors, and explore the energy-efficient and reliable RF. A straightforward approach to protect RF is building the pure STT-RAM based registers, named as all STT-RAM in this paper. We investigate a large number of GPGPUs benchmarks (detailed experiment methodologies are in Section 4.3), and observe that all STT-RAM degrades the performance by around 70% because the long-latency STT-RAM register writes cause pipeline stalls. Even worse, all STT-RAM increases the dynamic power consumption remarkably as a large voltage is requested during the STT-RAM write operation. The overall energy increases about 14% even though the leakage energy is substantially reduced by using STT-RAM. Relaxing the data-retention time to reduce the STT-RAM write delay and energy has been proposed in both CPU [40, 41] and GPUs [35]; however, it hurts the STT-RAM's soft-error robustness [43] and is not applicable to our study. We explore two soft-error protection mechanisms that are energy efficient and performance friendly in Section 4.2.1 and 4.2.2.

4.2.1 Lifetime-Aware LESS

4.2.1.1 Observation: The Lifetime of GPGPUs Register Values vs. RF Soft-Error Vulnerability

Previous investigations on GPGPUs workloads have shown that majority register values have short lifetime [20, 44]. Lifetime is defined as the number of instructions between the register value write and its last read. Note that each physical register is written with different values multiple times during the program execution, and we consider the lifetime of each unique register value instead of the lifetime of each physical register. We further find that those short-lived register values have little impact on the RF soft-error vulnerability. Figure 4.1(a) presents the distribution of register values with different lifetime. As it shows, for instance, 80% of the register values are alive no more than 10 instructions. Interestingly, the remaining 20% long-lived register values contribute to about 87% of the overall register soft-error vulnerability as shown in Figure 4.1(b). (The studied benchmarks and vulnerability estimation methodologies are introduced in Section 4.3.) Using STT-RAM to provide the shield for those long-lifetime register values would achieve near-full protection for the RF.

4.2.1.2 The Idea of Lifetime-Aware LESS and Its Implementation

We explore the hybrid SRAM and STT-RAM based RF, and propose lifetime-aware LESS to designate the long-lifetime (e.g., longer than 10 instructions) register values to STT-RAM while other short-lifetime register values to the SRAM-based RF. By doing this, the number of STT-RAM writes decreases dramatically, which effectively alleviates both the performance loss and dynamic power overhead compared to the all STT-RAM case. Meanwhile, lifetime-aware LESS still exhibits the strong capability in saving the leakage power due to the contribution of the STT-RAM based RF, and the overall RF power consumption drops substantially. Therefore, our technique optimizes both the RF soft-error robustness and power consumption.

Note that our proposed RF design is completely different from the register file caching (RFC) [20] and operand register file (ORF) [44] techniques, both introduced by Gebhart et al. The fundamental idea of RFC and ORF is using a small register file cache to keep the frequently read or soon to be read values and reduce the main RF access frequency for power saving. To further improve the soft-error robustness, one can keep the SRAM based implementation of the

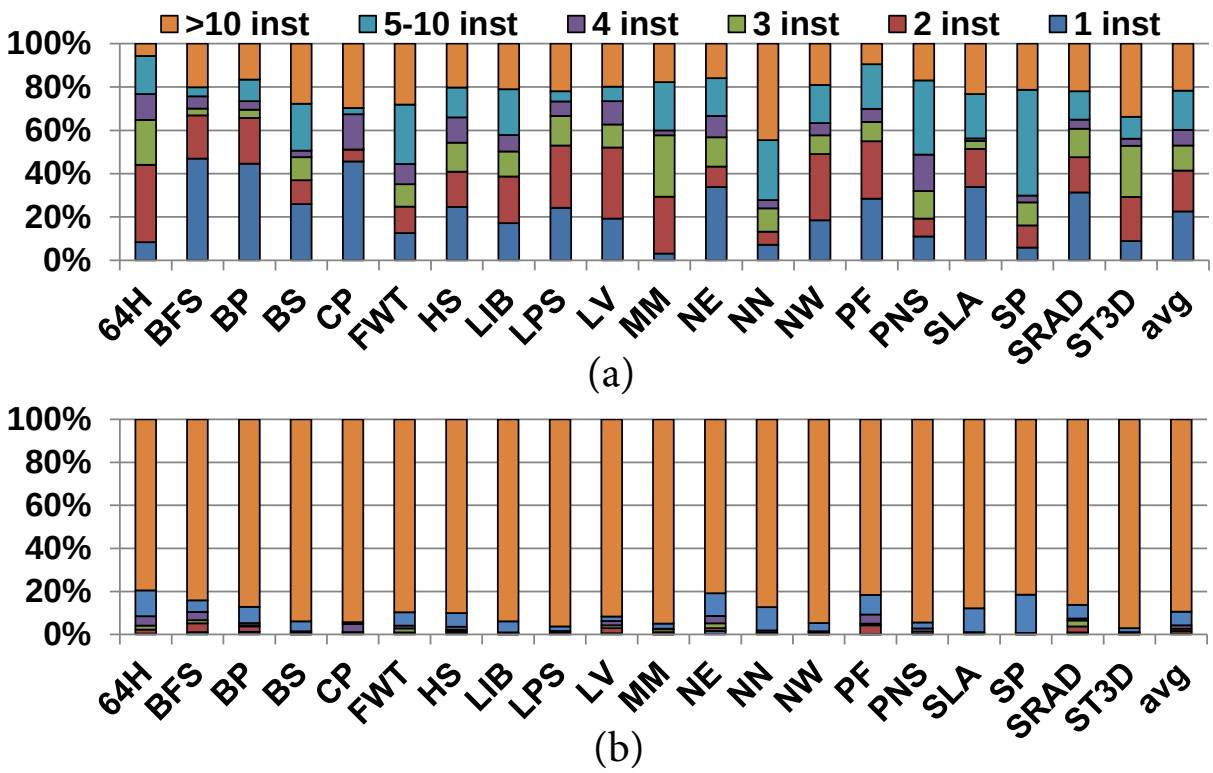

Figure 4.1: (a) The distribution of register values with different lifetime in instructions. (b) The contribution of register values in each lifetime category to the overall registers soft-error vulnerability.

small register file cache, while using STT-RAM to build the main RF. We name it as enhanced operand register file (E-ORF). However, no matter if the value is long or short lived, its first read can occur immediately once it is produced. Therefore, E-ORF does not have the capability to differentiate the value lifetime, not mention to designate register values with different lifetime into different RF segments to maximize the fault coverage with minimal STT-RAM writes. We evaluate E-ORF and compare it with our lifetime-aware LESS in section 4.4.1.2.

In lifetime-aware LESS, the lifetime of register values can be obtained with the help of the compiler through live variable analysis [20, 45]: the compiler analyzes the data flow graph in backward order and thus figure out the number of instructions between the register value's write and its last read. Furthermore, the register namespace is partitioned into two sets of architectural register names representing the SRAM and STT-RAM based RF, respectively. The long-lifetime (short-lifetime) register values will be properly mapped to the STT-RAM (SRAM) registers during the compilation time. The amount of SRAM and STT-RAM registers

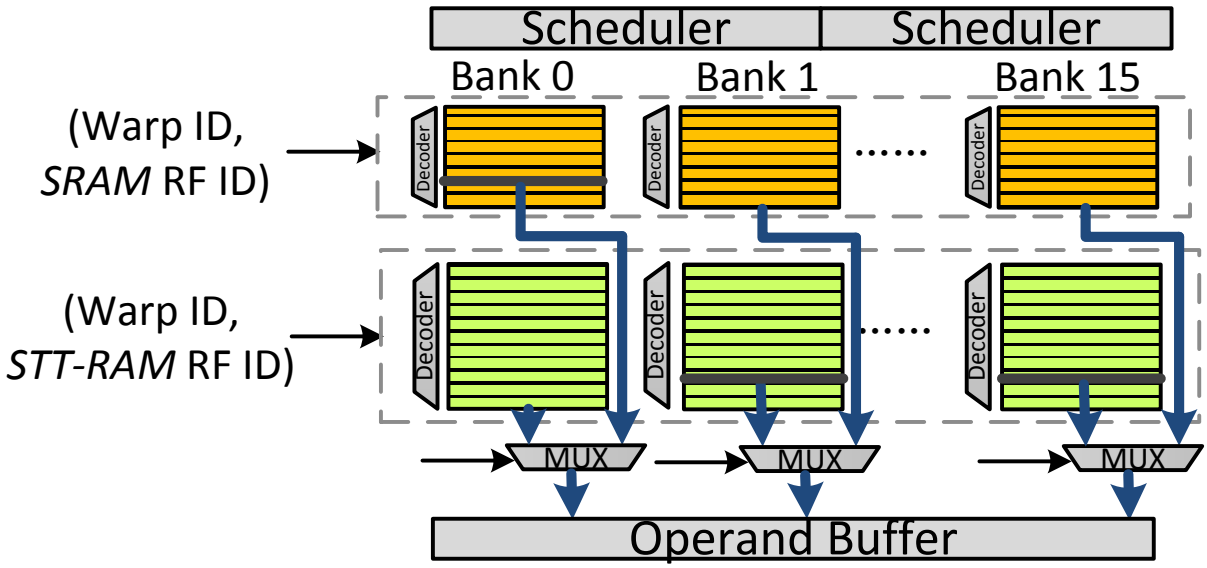

Figure 4.2: The SRAM (top) and STT-RAM (bottom) hybrid register file in lifetime-aware LESS

required by each thread will be sent to the GPUs during the kernel launch time.

Figure 4.2 shows the architectural design of our hybrid RF. In the default RF design, the warp ID is combined with the RF ID and the number of registers per thread to index the bank and the entry holding the register value. To keep the same register mapping mechanism, the SRAM and STT-RAM RF each has 16 banks with 1 read and 1 write ports, banks with the same bank number from the two RF segments share the data bus linked to the operand buffer and the execution units. As Figure 4.2 shows, the register index is based on the RF ID in certain name space (i.e., SRAM or STT-RAM) and the number of that kind of registers required by each thread. And one multiplexer is used to choose SRAM or STT-RAM bank for one register access. There is no other hardware required to support the STT-RAM or SRAM register accesses.

The SRAM and STT-RAM RF size partition largely impacts the effectiveness of lifetime-aware LESS, and an ideal partition will match the ratio between the maximum requirements of the two RF types through the entire execution time. We collect the ratio between the numbers of per-thread SRAM and STT-RAM register requirements provided by the compiler in various GPGPU benchmarks. The averaged ratio across all benchmarks is 75% STT-RAM and 25% SRAM when defining the register value lifetime that is longer than 10 instructions as long-lived

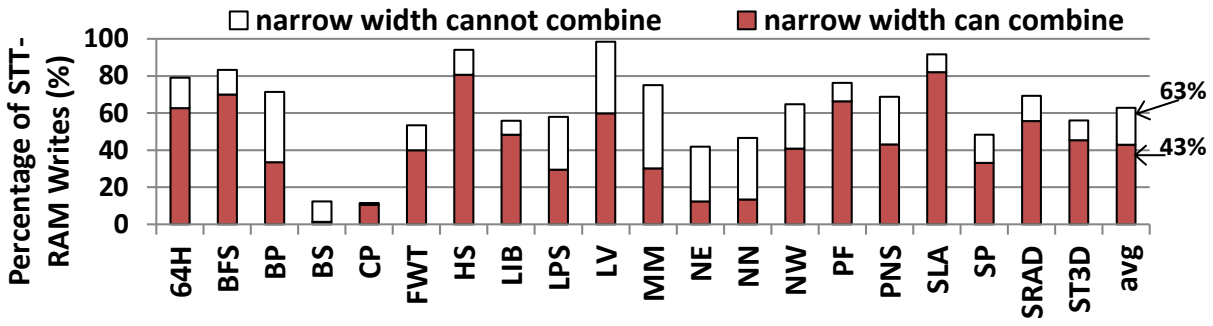

Figure 4.3: Percentage of STT-RAM writes that are narrow width and can be combined, and that are narrow width but cannot be combined.

register values (Based on our sensitivity analysis, setting the lower limit of the long lifetime as 10 instructions leads to the optimal results). We then use this ratio to partition 128KB RF in a default RF configuration into 96KB STT-RAM and 32KB SRAM RF segments. In the case that there are insufficient SRAM registers for threads in an SM, some free STT-RAM registers will be used to hold register values which are originally mapped to the SRAM RF, and vice versa.

In lifetime-aware LESS, the write operations to STT-RAM registers cannot be fully eliminated, and the 20% long-lifetime register value writes into the STT-RAM could still cause considerable performance loss. Thus, we further explore to absorb the STT-RAM write delay in the following subsection.

4.2.2 Narrow-Width-Aware LESS

4.2.2.1 Observation: Narrow-Width Values in STT-RAM Writes

Generally, not all register values will be 32-bit long in general-purpose applications [46]. In this study, a narrow-width write is defined as the case that the upper 16 bits of all the 32 same-named register values are zeros during the writeback stage. There are numerous narrow-width writes in GPGPUs applications [47], and we observe that 63% of the STT-RAM writes are narrow-width writes on average across the investigated benchmarks, as shown in Figure 4.3. Since a narrow-width write only requires half of the result bus bandwidth, two narrow-width STT-RAM writes can be combined to share the bus. This will greatly reduce the performance loss caused by STT-RAM writes. To keep the design simple, we only consider combining narrow-width STT-RAM writes from two warps that (i) occur in two continuous cycles, and

(ii) designate to different banks as there is only one write port in each bank. Note that the two warps can execute totally different instructions since the combination only happen at the write back stage. We observe that there are still 43% STT-RAM writes satisfying those two requirements, as shown in Figure 4.3. This is because threads in a kernel all execute the same code and warps usually proceed at similar rate under the fine-grained multithreading in the SM. Moreover, the same instruction code tends to write data with the same data width. Thus two continuous warps tend to both perform narrow-width writes. In summary, there are considerable amount of STT-RAM writes that can be combined to further improve the performance on top of lifetime-aware LESS.

4.2.2.2 The Idea of Narrow-Width-Aware LESS and Its Implementation

We propose narrow-width-aware LESS, which intelligently assigns the free data bus resources induced by the narrow-width STT-RAM write from one warp to another warp's narrow-width STT-RAM write during the writeback stage. Therefore, the result bus resources are well utilized to tolerate the negative performance impact caused by the long STT-RAM writes, and meanwhile, the error coverage obtained in lifetime-aware LESS is still maintained. Our proposed idea is orthogonal to the traditional write buffer design [48], which attaches a SRAM based write buffer to the STT-RAM bank to hide the STT-RAM write latency. Note that all narrow-width writes will proceed as long as there are available bus resources. In other words, the previous narrow-width write will not wait to combine with the following narrow-width write.

Figure 4.4 describes the implementation of the proposed technique at the writeback stage. We leverage the zero detection logic in execution units to detect the narrow-width values [16], and add one bit (i.e., narrow bit) to indicate whether all the 32 same-named register values are narrow-width or not. Since two narrow-width STT-RAM writes can be performed simultaneously, we need to keep the destination register IDs for both warps. They are named as high RF ID and low RF ID, respectively, as shown in Figure 4.4. The high (low) RF ID keeps the destination RF ID for data using the higher (lower) 16 bits of each 32-bit data slot in the result bus. In addition, two narrow-width valid bits (i.e., high-NV, and low-NV) are attached to those two destination register IDs to indicate if the higher/lower part of each 32-bit data slot contains narrow-width value. As Figure 4.4 shows, a tri-state logic and a demultiplexer are also used to

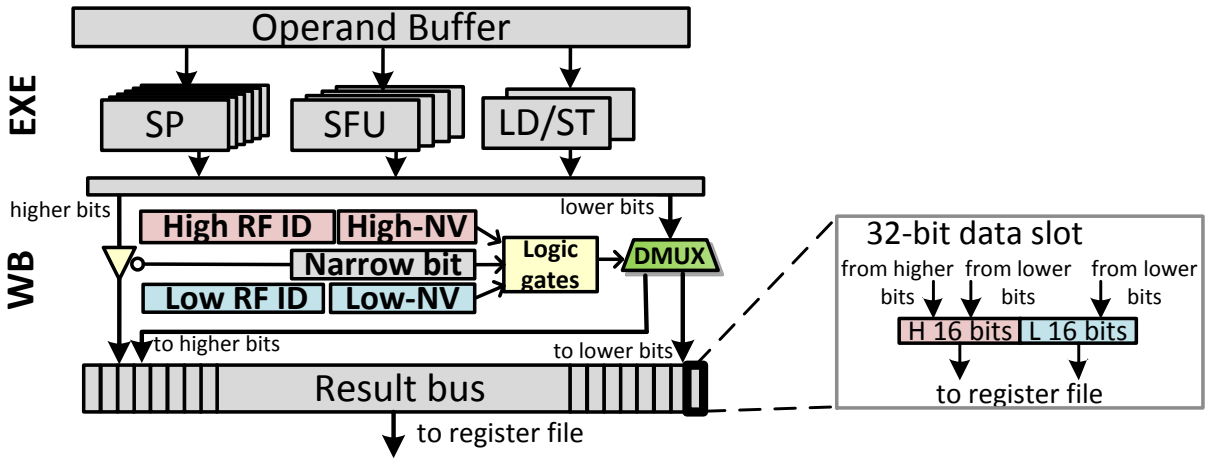

Figure 4.4: The implementation of narrow-width-aware LESS.

control the writes of the higher and lower 16 bits of each 32 same-named register values into the result bus, respectively.

When a warp instruction finishes execution, its narrow bit is checked. If it is set as 0, it means that values to be written back to the RF are normal-width. Thus both higher and lower 16 bits of each value will be written into the corresponding slots in the result bus when it is free. Moreover, the low RF ID is set as the instruction's destination RF ID, and both low-NV and high-NV are reset to 0, indicating this is a normal-width (i.e., 32-bit) write. When the narrow bit is set as 1, both high-NV and low-NV are checked, and there are three possible scenarios as follows: (i) if they are both 0 and the result bus is available, it implies that no previous write is occupying the result bus, and the lower 16 bits of each register value are directed to the lower half of each 32-bit data slot; (ii) in the case that only one narrow valid bit is 0, it implies that there is one previous narrow-width write currently using the bus, and the current narrow-width write will be designated to the free resources according to the narrow valid bit. In the above two scenarios, the corresponding destination register ID and narrow valid bit will be updated to record the information of the current narrow-width write. Note that the higher 16 bits of each value are disabled via the tri-state logic as they are all zeros; (iii) when both narrow valid bits are 1s, it means that the result bus is already fully occupied by two previous narrow width writes, thus the current narrow-width write has to be stalled and wait for the available bus resources. Finally, when writing data to the STT-RAM banks, the destination RF IDs combined with the narrow valid bits are used to write each half of the 32-bit data into

Table 4.1: Access time and energy for STT-RAM and SRAM based RF. Our hybrid RF in LESS is composed of 96KB STT-RAM and 32KB SRAM RF segments as discussed in Section 4.2.1.2

	SRAM based RF		**STT-RAM based RF**	
	32KB	***128KB***	***96KB***	***128KB***
Write Lat (ns)	0.497	1.06	5.124	6.036
Write Lat (cycles)	1	1	4	4
Read Dyn. Eng (nJ)	0.049	0.131	0.082	0.092
Write Dyn. Eng (nJ)	0.043	0.123	0.529	0.645
Leakage power (mW)	31.2	130	3.21	4.283

the appropriate banks. Once one warp instruction's write finishes, its information saved in the destination RF ID and narrow valid bits is cleared.

4.2.3 LESS: Putting It All Together

The merge of LA-LESS and NWA-LESS delivers a unified low-power and error protection scheme, LESS, for register file. Although some simple logic gates are introduced by LESS, it does not introduce additional area overhead thanks to the high-density of STT-RAM based RF, our gate-level modeling also proves this. Note that LESS aims to use STT-RAM to provide shield for registers, and the RF soft errors do not migrate to any other GPU structures.

4.3 Experimental Setup

We use GPGPU-Sim (v3.1.0) [19] and simulate PTXPlus instruction set, which is a one-to-one mapping to the native hardware instruction set. We focus on the 40nm process technology which is adopted in recently released GPU products. Our baseline GPU configuration models the Nvidia Fermi style architecture: the warp size is 32; the GPUs contain 16 SMs; each SM supports 1536 threads and 8 blocks at most; Each SM contains 128KB RF, 48KB shared memory, 16 KB L1 data cache, and 128KB L2 data cache. The scheduler applies the round robin scheduling policy. We collect workloads from Nvidia CUDA SDK [31], Rodinia Benchmark [32], and Parboil Benchmark [33]. We simulate all kernels in each benchmark to complete.

We use error coverage to evaluate the soft-error reliability of GPUs RF, which is calculated based on architecture vulnerability factor (AVF) [29]. A hardware structure's AVF refers to the probability that a soft error in that hardware structure will result in incorrect program results. The sum of AVF and error coverage is equal to one. As discussed in Section 4.1, STT-RAM is

considered as soft-error free in our experiment. Thus the error coverage for STT-RAM based register file is 100%. We build our power model based on the energy analysis tool CACTI [49]. The SM frequency is 600 MHz, and the supply voltage is 0.9V. We obtain the access time and energy of the SRAM and STT-RAM based RF with various size designs (listed in Table 4.1) based on a modified NVSim [50] simulator. In the experiment, the read access latency for SRAM and STT-RAM are both 1 cycle. Note that one STT-RAM write causes 4-cycle delay in our study, which is usually tens of cycles in previous studies on STT-RAM based L1/L2 caches in CPUs [40]. It is because SM runs at much lower frequency than CPU processors. The number of read/write operations to the two RF segments, and the total execution time are collected from the modified GPGPU-Sim to evaluate both RF dynamic and leakage energy. Our energy estimation is consistent with previous studies [20, 44, 17]. The energy consumed by the hardware introduced by LESS is also included in the model.

4.4 Evaluation

4.4.1 Lifetime-Aware LESS (LA-LESS)

4.4.1.1 The Effectiveness of LA-LESS

Figure 4.5 shows the normalized (a) execution time, (b) RF error coverage, and (c) RF energy (including both dynamic and leakage) obtained by LA-LESS when running the investigated benchmarks. The results are normalized to the baseline case without any optimization.

As it shows, on average, LA-LESS achieves the error coverage as high as 86% since the major RF vulnerability contributors (i.e., long-lived register values) are well protected by STT-RAM. Excitingly, LA-LESS gains 60% energy reduction on average, because the extra write energy caused by the infrequent STT-RAM RF writes is trivial when compared with the energy savings brought by the reduced per access energy on the small SRAM RF and the extremely low leakage on the STT-RAM RF.

As its main drawback, LA-LESS causes around 11% performance loss. Take NN as an example, the execution time increases by 60% because the long-lifetime register values account for 40% of the register values, and the frequent STT-RAM writes hurt the throughput. Fortunately, the energy savings for NN is promising because its warps contain few threads, the RF utilization in NN is low and leakage is the main component of the total energy. The reduced leakage in

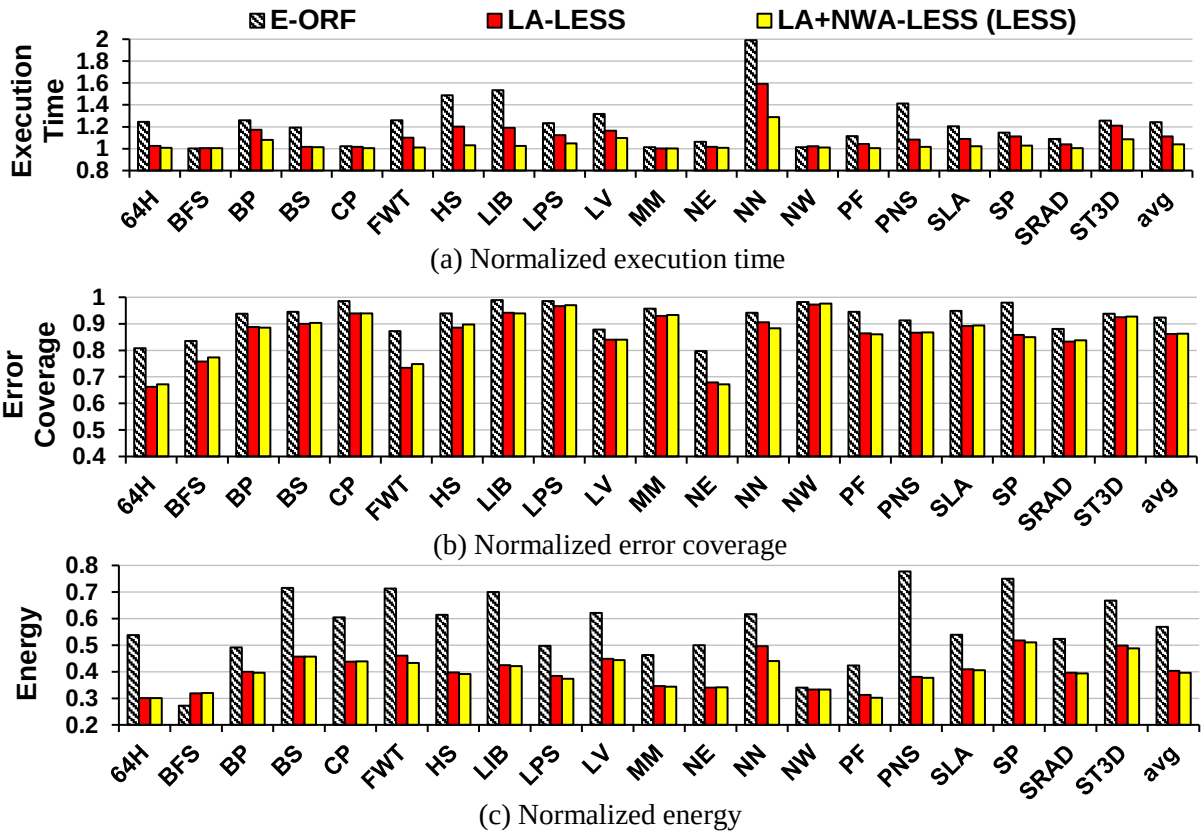

Figure 4.5: The normalized (a) execution time, (b) error coverage, and (c) energy of enhanced operand register file (E-ORF), the proposed lifetime-ware LESS (LA-LESS), and LA-LESS + the proposed narrow-width-aware LESS (LA+NWA-LESS).

the hybrid RF far outweighs the considerable STT-RAM write energy. ST3D exhibits similar behavior as NN.

4.4.1.2 Comparison with Enhanced Operand Register File (E-ORF)

Figure 4.5 also compares LA-LESS with the enhanced operand register file (E-ORF) technique that applies STT-RAM to build the main RF and uses SRAM to build the ORF (as discussed in Section 4.2.1.2). We follow the ORF configuration presented in [44] and set its size as 18KB. As it shows, LA-LESS achieves slightly (i.e., 6%) lower error coverage than E-ORF. In LA-LESS, the short-lived register values are always directed to the SRAM based RF. But in E-ORF, some short-lived register values whose first read occurs a few instructions away from the value generation will be recognized as exhibiting weak temporal locality and directed to the STT-RAM based main RF. The improved error coverage obtained by E-ORF is mainly from the additional protection for that kind of short-lived register values. As the serious negative impact

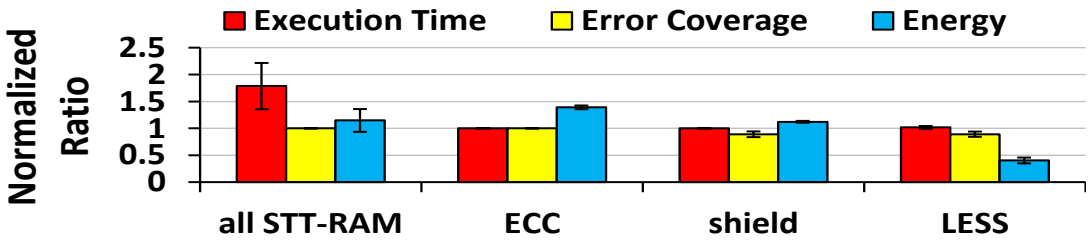

Figure 4.6: Comparison among all STT-RAM, ECC, shield, and LESS techniques, the standard deviations across benchmarks are also shown.

of gaining such small reliability improvement, ORF consumes 17% more energy and extends the execution time by 13% compared to LA-LESS because of the increased write operations to the STT-RAM based main RF.

4.4.2 LESS: Combining LA-LESS and Narrow-Width-Aware LESS (NWA-LESS) Together

As we described in Section 4.2, NWA-LESS can optimize 43% of the STT-RAM writes in LA-LESS. Instead of evaluating NWA-LESS technique separately, we focus on the aggregated impact and evaluate the effectiveness of LESS, which combines LA-LESS and NWA-LESS together, as shown in Figure 4.5. It is also named as LA+NWA LESS in Figure 4.5. Thanks to NWA-LESS, LESS further reduces the performance penalty when compared with LA-LESS and meanwhile, it maintains the high error coverage and low energy consumption. The performance benefit of NWA-LESS is obvious in benchmarks that contain numerous narrow-width STT-RAM writes. For example, the performance loss of ST3D reduces from 19% to 8%. This is because 45% of the STT-RAM writes are narrow-width values. On average, LESS performs well in almost all the benchmarks, and it obtains 86% RF error coverage, 60% RF energy reduction, with only 4% performance loss.

4.4.3 Comparing LESS with Other Soft-Error Protection Techniques for GPU RF

We further compare LESS with several other RF protection mechanisms in Figure 4.6: all STT-RAM that builds the pure STT-RAM based RF to achieve full error protection; ECC [51] that attaches the SEC-DED ECC (7-bit long) to every register for the error protection; Shield [52] that explores an ECC table to only protect the long-lifetime register values. As it shows, all STT-RAM has the highest (70%) performance loss and consumes 14% energy overhead when

writing every value into the pure STT-RAM based RF. ECC has strong capability in gaining full coverage without hurting performance, however, it needs 38% extra energy to keep the ECC bits, and generate/check ECC for each register access. Shield significantly reduces the ECC's energy overhead by 30% as it uses a relatively small ECC table to protect long-lived register values and achieve good error coverage. As can be seen, all these techniques introduce extra energy overhead. Excitingly, LESS saves tremendous amount of energy with negligible performance penalty. Moreover, the energy overhead increases in ECC related mechanism when the multi-bit error tolerance is desired; while LESS still keeps its benefit since STT-RAM is immune to soft errors. Thus, LESS is more efficient than ECC related techniques with regard to the reliability-energy co-optimization on GPUs RF.

We further use McPAT [53] to estimate the energy consumption of GPUs, and find that register file consumes 14% of the total GPUs chip energy (including both dynamic and leakage). Therefore, LESS achieves 8.4% energy savings for the entire GPUs chip; while ECC technique increases the chip's energy consumption by 5.3%. The energy-efficiency of LESS outperforms ECC even it causes 4% performance loss.

4.4.4 Die Cost Analysis

Building STT-RAM based RF in GPUs chip requires additional manufacturing cost. In this subsection, we estimate the die cost of our LESS technique. Eq. 4.1 describes the estimation model [54] used in this study:

$$die\_cost = \frac{wafer\_cost}{\frac{wafer\_area}{die\_area} \times yield}, \tag{4.1}$$

where yield is the good die percentage in a wafer. LESS increases the wafer cost by 5% due to the additional manufacturing process of STT-RAM [54]. We calculate the RF area via using McPAT [53], and find that it occupies around 13% of the GPUs die area. Since LESS reduces the RF area by 55% compared to the baseline case, it reduces the overall GPUs die area by 7%. Given the above information about wafer cost and die area, we find that the die cost of GPUs integrated with LESS is still slightly (e.g., 2%) lower than that of the baseline case under various yields ranging from 40% to 100%. Therefore, the additional manufacturing cost introduced by LESS does not have negative impact on the die cost.

Chapter 5

Mitigating the Susceptibility of Register File to Process Variations

GPUs support a great number of parallel threads and implement a zero overhead context switch among threads to hide the long latency operations. This requires an extremely large register file (RF) to keep the states and contents of all active threads. For instance, the register file size is 2MB in NVIDIA Fermi [51] and 6MB in AMD Cayman [55]. Such a large register file includes numerous parallel critical paths, and is quite sensitive to process variations. Even worse, to afford a greater number of threads executing simultaneously in the SM, the register file size continuously increases in recent GPUs product generations [51, 56]. Therefore, register file becomes one of the major units that affect the frequency and performance [35], and it is crucial to mitigate its PV impact.

The unique highly-banked register architecture design in GPUs provides a promising direction to efficiently tolerate the PV effect. However, generic PV mitigation techniques, such as adaptive body biasing (ABB) [57] and gate sizing [58], fail to exploit this unique feature and could cause considerable power and area overhead when directly applied to GPUs register file. In this paper, we propose to characterize and further leverage this highly-banked architecture feature to effectively mitigate the susceptibility of GPUs register file to process variations. Note that we assume other PV-sensitive structures, such as streaming processors, are handled by conventional PV tolerance mechanisms (e.g., ABB).

In this study, we first develop a novel fast and slow register classification mechanism to maximize the frequency improvement in the highly-banked register architecture. We then exploit the unique features in GPGPU applications to intelligently tolerate the extra access delay to the slow registers. The contributions of this work are as follows:

- We observe that PV exhibits much stronger systematic effects in the vertical direction than that in the horizontal direction within each RF bank in the state-of-the-art GPU register file floorplan [16, 17]. We then propose a coarse-grain register classification mechanism by vertically dividing each RF bank into sub-banks, and applying the variable-latency technique at the sub-bank level (VL-SB). VL-SB is able to attain the same frequency improvement as the fine-grain classification at the register level.

- We further propose RF bank re-organization (RF-BRO) to virtually combine sub-banks with the same speed type (i.e., fast, and slow). Therefore, the same-named registers in a RF bank entry share the uniform access delay, and the newly formed RF banks can be classified into fast and slow categories. We also show that the proposed VL-SB and RF-BRO techniques are applicable to other different register file architecture design [20, 44].

- In order to mitigate the IPC loss caused by the slow RF banks access under VL-SB+RF-BRO, we propose to grant warps that heavily use fast RF banks a higher issue priority (in GPUs, threads are executed in warps). It forces fast RF banks to serve more threads and minimizes the use of slow RF banks, and also appropriately enlarges the progress difference among warps to effectively hide stalls caused by the long-latency operations.

- We finally propose to virtually build multiple hybrid RF banks: each is composed of both fast and slow sub-banks. And only their fast sub-bank portions are enabled to hold registers for active threads in partially active warps. Therefore, their unused slow sub-bank portions have no impact on the RF access delay.

By combining all the explored techniques together, we achieve 15% frequency improvement compared to the baseline case without any optimization under PV, and 24% IPC improvement compared to the fine-grain register classification mechanism (i.e., VL-RF technique).

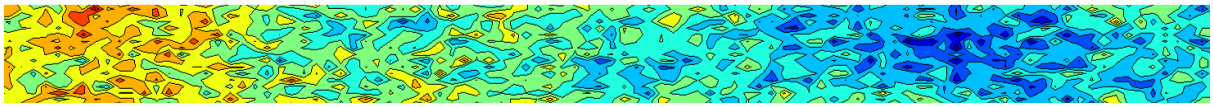

Figure 5.1: V_{th} variation map of GPUs regsiter file.

5.1 Frequency Optimization for GPUs Register File under Process Variations

5.1.1 Modeling PV Impact on GPUs Register File

Process variations (PV) are a combination of random effects (e.g., due to random dopant fluctuations) and systematic effects (e.g., due to lithographic lens aberrations) that occur during transistor manufacturing. Random variations refer to random fluctuations in parameters from die to die and device to device. Systematic variations refer to layout-dependent variations which cause nearby devices to share similar parameters. Die-to-die (D2D) variations mainly exhibit as random variations, and Within-die (WID) variations consist of both random and systematic variations. We focus on WID variations since D2D effect can be modeled as an offset value to all the devices in the chip. Among the design parameters, effective channel length (L_{eff}) and threshold voltage (V_{th}) are two key parameters subject to large variations [59]. The high V_{th} and L_{eff} variations cause high variations in transistor switching speed.

Recently, an architectural model of process variations, VARIUS [59] has been developed to quantitatively characterize the frequency variation in CPUs. In this study, we leverage the SRAM timing error model in VARIUS, and modify it to model the PV effects on GPUs register file. We focus on the 32nm process technology that is generally used in the state-of-the-art GPUs [19]. We set the WID correlation distance coefficient as 0.5, and assume V_{th}'s σ/μ = 12%, L_{eff}'s σ/μ = 6% [59, 60], the random and systematic components have equal variances for both V_{th} and L_{eff} [61, 59, 60]. Figure 5.1 shows an example of V_{th} variation map for GPUs register file. Note that we also perform the sensitivity analysis by varying the ratio between the random and systematic components when evaluating our proposed techniques in Section 5.4.3. We generate 100 chips for statistical analysis, and present the averaged result.

Note that each SM in GPUs exhibits different FMAX under PV. We model SM-to-SM variations as the ratio of frequencies of the fastest and the slowest SM in a GPUs chip, and

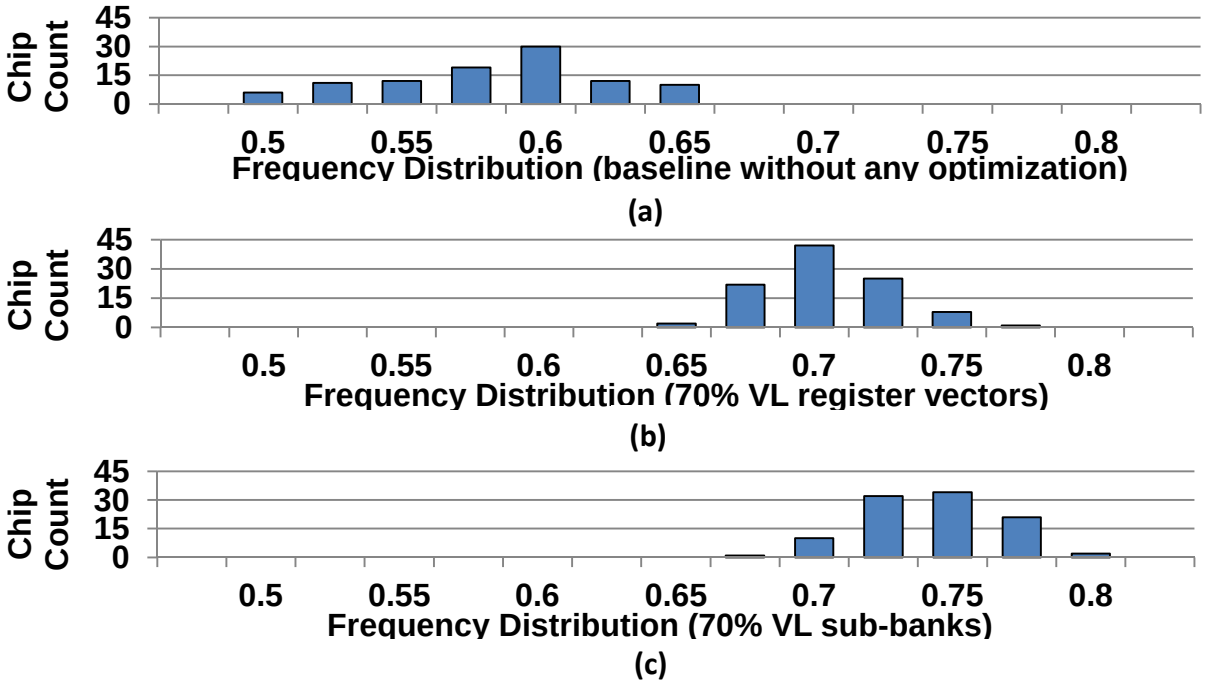

Figure 5.2: Register file frequency distribution under process variations.

model the within-SM variations as the ratio of frequencies of the fastest and slowest critical path. Based on our experimental results, within-SM variations are 1.7 which is larger than SM-to-SM variations that are around 1.3. This is because each SM has numerous parallel critical paths; while there are only tens of SMs. SM-to-SM variations have been smoothed out as FMAX of each SM is determined by the slowest critical path in it. Moreover, there have been techniques letting each SM in GPUs run at their own FMAX for PV mitigation [62]. We thus perform the variability analysis within the SM. As mentioned before, register file is one of the major structures that limit the SM frequency, and we assume other PV-sensitive structures are handled by conventional PV tolerance mechanisms. Thus, the SM frequency is determined by the register file frequency which is the reciprocal of the slowest register access time. Figure 5.2(a) demonstrates the register file frequency distribution over 100 chips. There are 15 SMs in the Fermi architecture, and we use the averaged register file frequency across SMs to represent the frequency for one chip. Therefore, Figure 5.2(a) mainly shows the impact of within-SM variations on frequency. As it shows, the mean frequency degradation is 40% compared to GPUs without PV. This is because numerous critical paths in GPUs register file lead to the large within-SM variation which significantly decreases the frequency.

5.1.2 Variable-Latency Sub-Banks (VL-SB) in GPUs Register File

5.1.2.1 Extending VL-RF to GPUs RF

There have been several PV tolerant techniques explored to optimize the multi-ported register file in CPUs [63, 64]. For instance, Liang et al. [63] proposed n% variable-latency RF (n% VL-RF) to partition all registers read ports into fast and slow categories. The slowest (100-n)% ports are marked as slow and accessed in two cycles. They are not considered in determining the frequency so that the chip frequency increases in the presence of PV. When a slow port is assigned to read a register, port switching technique is triggered to switch to a fast port attached to the same register and avoid the extra cycle delay. However, port switching is not applicable to modern GPUs register file. This is because implementing multiple ports to a sizeable register file in GPUs is not practical, and the highly-banked register architecture has become a widely accepted design to provide the high register access bandwidth [16, 17, 18, 19]. Although the VL technique can be simply extended to GPUs registers by exploring the fast and slow registers based on their access delay to attain the optimal frequency improvement, such a fine-grain register classification causes extremely high IPC degradation. This is because multiple (e.g., 32) same-named registers are accessed simultaneously in GPUs, and the access latency increases as long as one slow register gets involved.

In our baseline RF design, each SM is equipped with 128KB register file that is composed of 32K 32-bit registers. In order to reduce the impact of the extremely slow register and boost the frequency under PV, one can apply the n% variable-latency RF technique [63] to divide those 32K registers contained in the SM into fast and slow categories depending on their access delay. Based on our sensitivity analysis, setting n% as 70% delivers the optimal trade-off between frequency and the amount of slow registers. We implement the 70% variable-latency register file (70% VL-RF): the slowest 30% registers are classified into the slow category and will take 2 cycles to finish the read/write operation; frequency is determined by the slowest register of the remaining 70% registers in the fast category.

However, the variable-latency RF causes serious IPC loss. Recall that one operand access in an instruction warp involves the parallel accesses to 32 same-named registers from all threads within the warp. As long as there is one slow register among those 32 registers, the operand

access latency is 2 cycles. In our baseline RF design [16, 17], although the 32 same-named registers are implemented close to each other and included in a single entry of the RF bank, the PV exhibits weak systematic effects for such a 1024-bit wide entry. As a result, most 32 same-named registers contain at least one slow register, and the operand access latency is generally extended. We observe 23% IPC degradation under 70% VL-RF compared to the baseline case without any optimization under PV. Moreover, the variable-latency technique requires an extra bit per register to record the speed information as fast or slow, leading to large power and area overhead.

An alternative design to avoid the large IPC degradation is to consider each 32 same-named registers as a group, called register vector, and apply the 70% variable-latency technique at the register vector level, namely, 70% VL-RV. Similar to VL-RF, VL-RV requires an extra bit per register vector and causes considerable power and area overhead. More importantly, there is large delay variability among registers within a register vector but the slowest one determines its access delay. In other words, the variations at register level are significantly smoothed out at the register vector level. Thus, dividing register vectors into fast and slow categories has limited effect on frequency optimization.

5.1.2.2 Variable-Latency Sub-Banks (VL-SB) in RF

In our baseline RF design [16, 17], each RF bank holds 64 1024-bit wide entries. Therefore, PV exhibits much stronger systematic effects in the vertical direction than that in the horizontal direction within each RF bank. In this study, we focus on this wide-entry RF architecture containing 16 banks, and the explored techniques perfectly fit to other RF architecture which is discussed in Section 5.1.5.

We propose n% variable-latency sub-banks in GPUs register file (named as VL-SB) that vertically divides each RF bank into several sub-banks, and registers within each sub-bank share the same access speed that is constrained by the slowest one. Sub-banks exhibit distinct access delay, and the slowest (100-n)% ones are marked as slow. There is small delay variability among registers contained in a sub-bank due to the systematic effects, therefore, the large variations at register level is well maintained at the sub-bank level in VL-SB, which maximizes the frequency improvement under the VL technique.

Note that both read and write delay is considered in VL-SB. We observe that sub-banks with long (short) read delay are highly likely to exhibit long (short) write delay under the impact of systematic variations. This makes the sub-banks classification quite straightforward, and leads to only two categories that are fast read+ fast write (i.e., fast sub-bank) and slow read + slow write (i.e., slow sub-bank). We choose to divide each RF bank into 2 sub-banks because further aggressively performing the fine-grain partition (i.e., 4 sub-banks or more) does not lead to an obvious frequency increase based on our sensitivity analysis. As can be seen, only 32 bits are required to keep the speed information for the 32 sub-banks in VL-SB, which causes negligible area overhead.

Since PV also causes delay variability among SMs in the same GPUs chip, one can change the value of $n\%$ during the sub-banks partition in different SMs to ensure the uniform frequency across SMs. In that case, each SM has distinct number of fast sub-banks which may affect the IPC. It is encouraged to employ the per-SM clocking as discussed in [62], we thus keep the uniform partition criterion (i.e., 70%) for each SM, and our techniques are orthogonal to the previously explored inter-SM level PV mitigation mechanisms [62].

Figure 5.2 (b) and (c) justify the effectiveness of 70% VL register vectors and 70% VL sub-banks by showing the RF frequency distribution when the two techniques are enabled, respectively. Every 32 nearby register vectors in VL-RV are grouped into an array to keep the same area overhead as VL-SB for a fair comparison. This makes 32 arrays in total, and the slowest one in the fastest 70% arrays decides the frequency. As Figure 5.2(b) demonstrates, the mean frequency in 70% VL-RV increases 10% compared to the baseline case presented in Figure 5.2(a), while 70% VL-SB in Figure 5.2(c) is able to boost the mean frequency by 15%.

Note that the structural redundancy technique [65], which adds redundant structures to the processor as spares, is not applicable to eliminate the slowest x% critical paths in GPUs RF for frequency boosting. If just applying redundancy to replace the slowest x% register vectors which may distribute across all the RF banks, the register mapping becomes extremely complicated and impractical. If applying redundancy to replace the slowest x% RF banks, it will cause considerable area overhead. Moreover, selective word-line voltage boosting [66], which was applied to reduce the cache line access latency under PV effect, is not applicable to GPUs

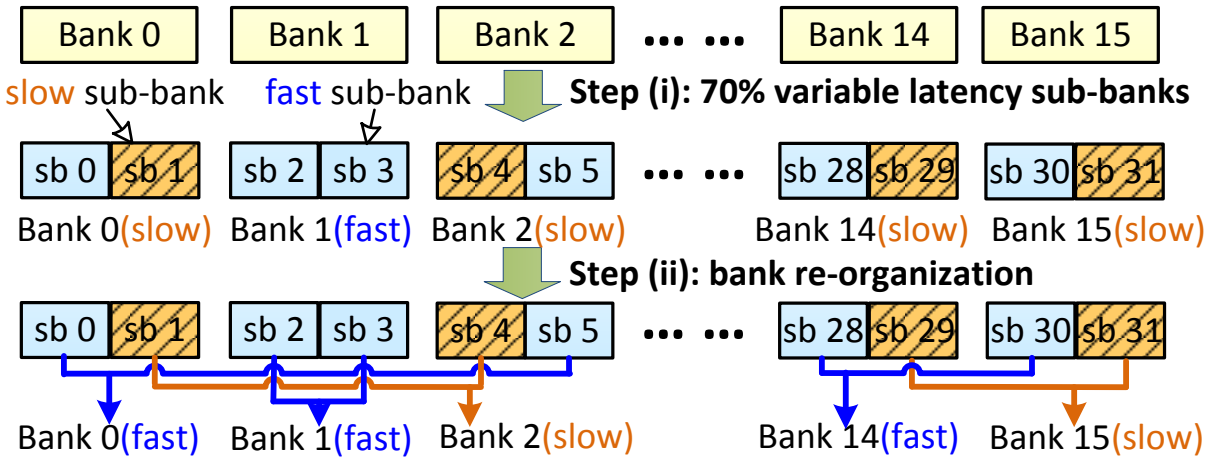

Figure 5.3: An example of applying 70% VL-SB (step i) and RF-BRO (step ii)(sb:sub-bank).

RF. Since RF is accessed more frequently than caches, selective word-line voltage boosting will cause considerable energy overhead to GPUs RF.

5.1.3 Register File Bank Re-Organization (RF-BRO)

The VL-SB technique faces the same challenge as VL-RF since it distributes registers belonging to the same register entry (i.e., register vector) into two sub-banks, which may be classified into different categories. We further propose RF bank re-organization (RF-BRO) on top of VL-SB that virtually combines two sub-banks from the same category to form a new RF bank. RF-BRO ensures the uniform access latency during the parallel accesses to 32 same-named registers. An operand access from an instruction warp is able to finish in 1 cycle as long as it is mapped to the newly formed RF bank that is composed of two fast sub-banks.

Figure 5.3 shows an example of applying 70% VL-SB with RF-BRO on GPUs RF. When VL-SB is applied (step i in Figure 5.3), the 16 RF banks are vertically divided into 32 sub-banks, and 22 of them are fast based on the 70% partition criterion (only 5 RF banks are shown in Figure 4 for the illustration purpose). However, 10 RF banks still need 2 cycles to finish the read/write operation because they all contain slow sub-banks. With the help of RF-BRO (step ii in Figure 5.3), sub-banks with the same type are virtually grouped to re-build the RF banks. For example, the new RF bank 0 is composed of sub-bank 0 and 5, and its access delay decreases to 1 cycle as it gets rid of the slow sub-bank 1. As a result, there are only 5 RF banks exhibiting 2-cycle access delay. In other words, the percentage of slow RF banks reduces from 62.5% to 31.25% under RF-BRO.

5.1.4 Implementation

We divide the word-line in each RF bank into two segments to obtain the sub-banks. This is a widely used method in SRAM-based structures to reduce the delay [67] or save dynamic power when a single word need to be accessed in a large cache [68]. Each word-line segment in the sub-bank is equipped with a local decoder in our VL-SB.

The implementation of VL-SB is similar to that of the VL-RF technique [63]: the speed information for each RF sub-bank will be collected by using BSIT [69] at chip test time. This information is used to mark each sub-bank as fast or slow, and also set an appropriate SM frequency. Note that the bank re-organization does not physically move any sub-bank during the chip fabrication. It virtually re-builds the RF banks by introducing a 16-entry bank organization table: each entry in the table records the IDs of two sub-banks that are assigned to the newly formed RF bank. In order to implement the bank re-organization technique, the IDs and the type (fast or slow) of every two same-type sub-banks are configured into a ROM at the chip test time. They will be loaded from the ROM and written into the SRAM-based bank organization table once GPUs are powered on. Based on our gate-level modeling, the access latency of bank organization table is negligible due to its small size, thus it does not increase the cycle time of the pipeline stage.

Figure 5.4 depicts the implementation of the two proposed techniques. During an operand access to the 32 same-named registers, the RF bank ID obtained from the warp and register IDs is used to index the bank organization table, and retrieve IDs and the speed type of corresponding sub-banks. The same entry in those two sub-banks will be activated simultaneously for operand access. For example, when RF bank 2 is accessed, sub-bank 1 and 4 are enabled as shown in Figure 5.4. Meanwhile, the speed type obtained from the table will be ANDed with a busy signal, and a slow type leads to a distribution of the signal to all sub-banks. Only sub-banks that are activated to fulfill this operand access will receive the busy signal and save it into the attached latch. This is used to prevent the pre-charge at next cycle to ensure the register read lasting for two cycles and finishing correctly. In GPUs RF, an arbitrator is applied to select a group of non-conflicting accesses and send to the RF banks at every cycle [16, 17, 19]. Therefore, the busy signal is also sent to the RF arbitrator to stall the following register read/write to the

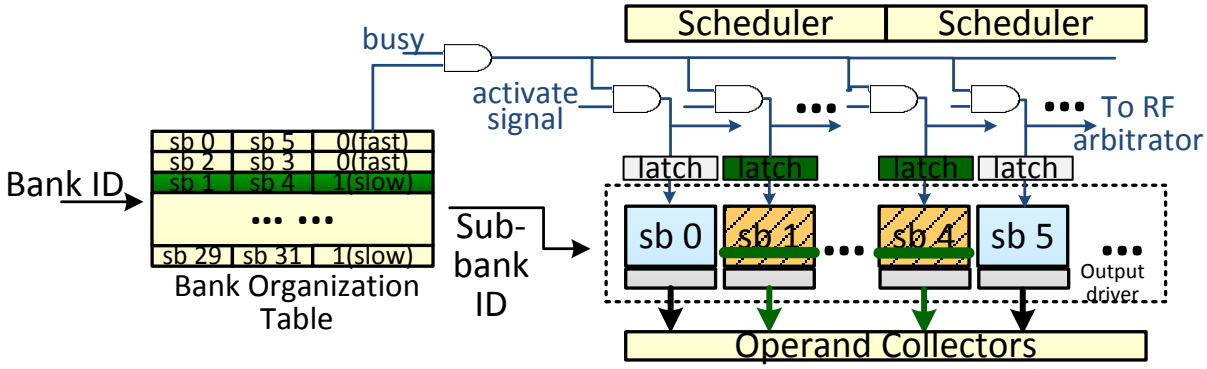

Figure 5.4: Hardware implementation of variable-latency sub-banks (VL-SB) and register file bank re-organization (RF-BFO).

same RF bank.

5.1.5 Feasibility in Alternative Register File Architecture

Alternative register file architectures are used in contemporary GPUs. One example is to group 4 SIMD lanes in an SM into a cluster, and 8 clusters form a complete 32-wide SM [20, 44]. In this case, each cluster contains 4 register banks, and each entry in a bank is only 16-byte wide that contains the register values for 4 threads in a warp (i.e., 4 same-named registers). Thus 32 same-named registers are distributed into 8 banks (one bank per cluster), and the same entries from 8 RF banks are accessed simultaneously for one operand access. As can be seen, the systematic effects for those 32 same-named registers are weak since they are even distributed to different clusters. Neither VL-RF nor VL-RV techniques could deliver good frequency improvement. We can consider this narrow-width style RF architecture as vertically dividing the 1024-bit wide register vectors (i.e., 32 same-named registers) into 8 sub-banks. Thus the proposed RF-BRO technique can be directly applied for the performance optimization under PV: we will not sub-divide each register bank, instead we adopt the VL technique to those 16-byte wide banks, and virtually re-organize 8 banks with the same speed to form the 32 same-named registers. In summary, our VL-SB RF design and the RF-BRO mechanism built upon it are applicable to other GPUs RF design.

5.2 Mitigating the IPC Degradation under VL-SB and RF-BRO

Although the VL-SB+RF-BRO technique explored in Section 5.1 largely optimizes the GPUs RF frequency under PV impacts, there are still about 30% slow banks among the virtually

re-organized RF banks, leading to around 9% IPC degradation based on our experimental results shown in Section 5.4.1. We further propose a set of techniques that harness the unique characteristics in GPGPUs applications to minimize the IPC loss. Note that the slow (fast) RF banks mentioned in this section are RF banks that are virtually composed of two slow (fast) sub-banks under RF-BRO.

5.2.1 Register Mapping under VL-SB and RF-BRO

The SM in Fermi style architecture is armed with two warp schedulers. Warps with odd and even IDs are dispatched into those two schedulers, respectively. At every cycle, two instruction warps are issued based on the round-robin scheduling policy, and they are likely to have identical PC since all threads in a kernel execute the same code. Mapping the same-ID registers from different warps into the same bank seriously exacerbates the bank conflicts, because different entries within a bank may be requested by the two simultaneously issued instruction warps. Therefore, the register-to-bank mapping mechanism follows the Eq. 5.1

$$(warp\_ID + register\_ID)\%(the\ number\ of\ banks), \tag{5.1}$$

to ensure that different banks hold the same-ID registers across the warps. As Eq. 5.1 shows, consecutive warps tend to map their same-ID registers into nearby RF banks. For instance, R1 from warp0 and warp1 are mapped to bank 1 and bank 2, respectively. Generally, consecutive warps exhibit strong data locality [22], their same-ID registers should be allocated to RF banks with same speed type to ensure they execute at similar progress. We propose to save IDs of same-type sub-banks into consecutive entries in the ROM at the chip test time, therefore, bank0-bank10 are fast while bank11-bank15 are slow in the bank organization table under VL-SB and RF-BRO techniques. By using Eq. 5.1, there are a number of registers per warp mapping to the slow RF banks. And the slow bank keeps registers with different IDs at the warp level. Figure 5.5 demonstrates an example of register mapping: R11-R15 from warp0 while R0-R4 from warp11 are assigned to the slow RF bank11-bank15.

5.2.2 Fast-Bank Aware Register Mapping

It has been observed that around 50% of registers are not even allocated by the compiler for the application execution [18]. This unique feature can be leveraged to minimize the use of slow RF banks by mapping registers to fast banks to the maximum degree during the kernel launch time. For the benchmarks that have high RF utilization, slow banks are used to ensure high level TLP. We find that a small set of registers have much higher access frequency than other registers allocated to the same warp, and they are usually the registers with small ID. For instance, each warp in benchmark BN (detailed experiment methodologies are in Section 5.3) is assigned 14 register vectors, and R0-R2 are used 250% more frequently than R3-R13. This is because the compiler tends to re-use the small-ID registers. We explore a novel fast-bank aware register mapping mechanism (named as FBA-RM) that consists of two steps: (1) obtaining the register resource requirements at the kernel launch time, and mapping registers only to fast banks if they are large enough to hold all registers needed by the parallel threads (i.e., reducing the number of RF banks to 11 in Eq. 5.1 during the mapping); (2) allocating the large-ID registers to slow RF banks when they have to be used, therefore, the slow banks are rarely accessed. In that case, Eq. 5.1 is used for small-ID registers to fast banks mapping and large-ID registers to slow banks mapping, respectively.

The major disadvantage of the FBA-RM technique is the increased bank conflicts as most RF accesses are limited to fast banks, and it fails to effectively mitigate the IPC degradation. In Section 5.4.2, we perform the detailed evaluation about FBA-RM by comparing it with other proposed techniques in the following subsections.

5.2.3 Fast-Warp Aware Scheduling (FWAS) Policy

Considering that modifying the register mapping mechanism to minimize the use of slow banks has little impact on performance optimization, we thus adopt the default mapping mechanism in Fermi and propose a set of methods to hide the extra access delay on slow banks. As Figure 5.5 shows, the frequently accessed small-ID registers in a number of warps (e.g., warp 11) are mapped to slow RF banks, which seriously delay their execution progress. And we define this kind of warp as slow warp. We further define warps whose frequent register accesses in fast banks as fast warps (e.g., warp 0). The execution progress for fast warps is delayed

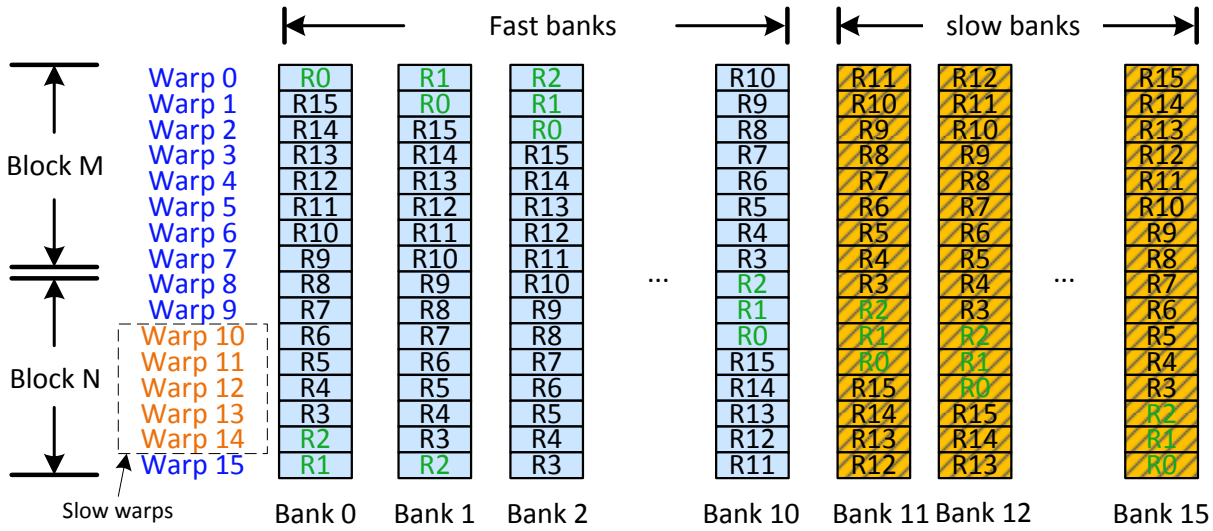

Figure 5.5: The idea of FWAS. R0-R2 are frequently accessed small-ID registers. Warp 10-14 in block N are slow warps.

somehow when the default warp scheduling policy (i.e., round-robin) is applied. This is because round-robin policy gives each warp the same issue priority, and when the slow warps occupy the pipeline resources (e.g., the issue and write slot), the ready fast warps cannot leverage those resources for execution. As a result, there is a small progress difference between fast and slow warps within the same SM.

We propose the fast-warp aware scheduling policy (named as FWAS) that assigns fast warps higher issue priority than slow warps to maximize the progress difference between them. This explores the unique opportunities to mitigate the IPC loss as follows:

(1) The fast warps have shorter execution time, thus the RF resources allocated to these warps are able to serve more warps during execution. This is effective for kernels including a large number of blocks that cannot be fully distributed to SMs at one time;

Figure 5.5 explains this opportunity in detail. Generally, there are multiple blocks executing concurrently in an SM. Figure 5.5 shows an example SM with two blocks: block M and block N, each block contains 8 warps. The frequently accessed registers (i.e., R0-R2) of all warps in block M (i.e., warp 0-7) are mapped to fast banks, so all warps belonging to block M are fast warps. On the contrary, in block N, most of their frequently accessed registers in warp 10-14 are mapped to slow banks, thus these warps are considered as slow warps. During program execution, FWAS prioritizes fast warps and allows them to finish earlier. As a result, warp

10-14 will left behind and become the bottleneck for block N. When all warps in block M finish execution, the new coming block within the same kernel will be assigned to take the resources (e.g., warp slots, registers) just released by block M. It also contains more fast warps (i.e., use those fast banks) because its warps will be assigned the same warp IDs as those warps in block M. On the other hand, block N will not release its resource until all its slow warps finish execution. As a result, the number of blocks that contain a larger amount of fast warps increases (in other words, fast banks are able to serve more warps) during the entire kernel execution, leading to the IPC improvement.

(2) The fast warps are able to start their off-chip memory accesses earlier, which alleviate the memory contention under the round-robin policy and reduce the pipeline stall time. This is quite effective in memory-intensive benchmarks.

In order to implement the FWAS policy, the fast warps have to be identified at the issue stage. However, there is no clear boundary between fast and slow warps, because the frequently accessed registers in a few warps are allocated to both fast and slow banks under the default mapping mechanism, for instance, the heavily used R0-R2 in warp9-warp10 in Figure 5.5. Although slow warps also access fast banks, it is highly possible that an instruction with fast bank access belongs to a fast warp. Instead of performing the accurate fast and slow warp identification, we choose to simply give the instruction warp that requires fast bank accesses higher issue priority. At the decode stage, an instruction warp is marked as fast if it has fast RF read/write. Note that only when all its operands reads are mapped to fast RF banks, an instruction warp is considered as possessing fast RF read. The one-bit fast/slow information combined with the ready bit is sent to the selection logic during the issue stage to perform the FWAS policy.

5.2.4 Hybrid RF Bank for Partially Active Warps

5.2.4.1 Observations on Partially Active Warps

In some GPGPUs benchmarks (e.g., NN), the block size is even smaller than the warp width of 32 threads, and warps do not contain enough active threads through the entire execution. Moreover, in benchmarks with heavy branch divergence, warps usually include several inactive threads. Both the two cases result in partially active warps, which will not use all the 32 registers contained in a register vector. We further find that a few register vectors usually have

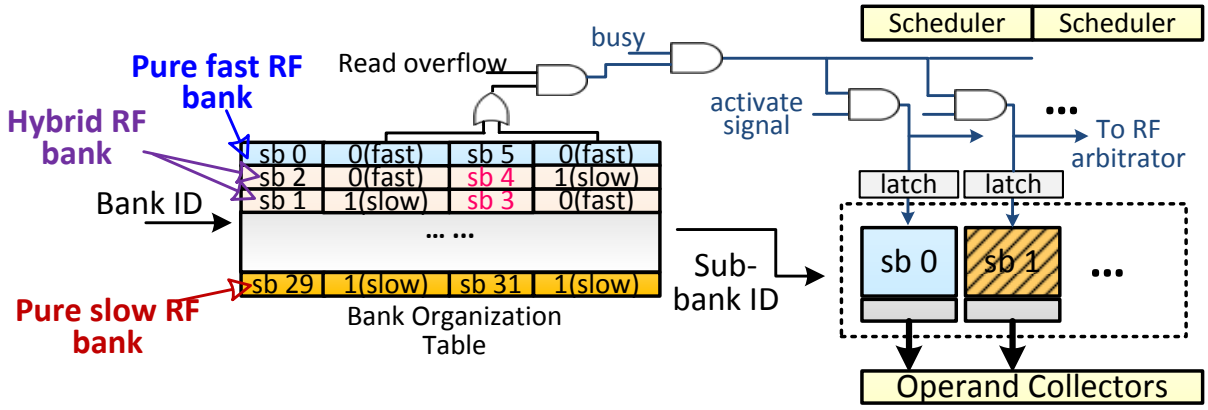

Figure 5.6: The implementation of virtually building hybrid RF banks based on RF-BRO technique.

the fixed registers utilized in a branch induced divergent warp. This is because a divergent branch that is depending on the programmatic values (e.g., block ID, thread ID) always causes the fixed threads in a warp active, and such programmatic-value dependent branches occur quite frequently in GPGPUs applications [70]. The partially utilized register vector does not need to map to the pure fast RF bank that is composed of fast sub-banks only. As long as its utilized registers are allocated to fast sub-banks, the register access operation latency can be constrained into one cycle.

5.2.4.2 The Idea of Hybrid RF Bank

In this study, we propose to build multiple hybrid RF banks and harness the above unique characteristics in GPGPUs benchmarks to further improve the performance. Different from the RF-BRO technique that always combines same-type sub-banks to organize a RF bank, each hybrid RF bank consists of one fast and one slow sub-bank. It is particularly used to hold register vectors with fixed registers active in partially active warps. Since only the fast sub-bank portion is required to conduct the read/write operations, the hybrid RF banks can be treated as fast banks, which increase the total number of fast banks without sacrificing the RF frequency and meanwhile, mitigating the IPC loss. Figure 5.6 depicts the idea of virtually building hybrid RF banks based on our RF-BRO technique. The new GPUs register architecture contains three types of RF banks: pure fast, pure slow, and hybrid categories. Note that aggressively increasing the hybrid RF size reduces the pure fast RF size since the overall amount of sub-banks remains

the same. This may force some fully occupied register vectors that are supposed to be mapped to pure fast RF banks being re-directed to the hybrid banks and using their slow-bank portion, thus, leading to the performance degradation. Ideally, the size of hybrid RF should match the total number of partially used register vectors. However, that number varies significantly across benchmarks.

The programmatic-value dependent branches can be detected at the compilation time using taint analysis [70]. We adopt this method to mark the special register vectors that prefer hybrid RF. The register namespace is partitioned into two sets of architectural register names representing the pure fast/slow and hybrid RF, respectively. The amount of those two types of registers per thread will be sent to the GPU during the kernel launch time, and meanwhile, the RF banks are re-configured to match the resource requirements on hybrid RF banks. To avoid the excessive hybrid RF resources, the number of hybrid RF banks is equal to the integer part of a product of the RF banks quantity (i.e., 16) and the ratio of requested hybrid RF size to the overall requested RF size. For benchmarks that have insufficient threads to fill up a warp (this can be identified at the kernel launch time), all slow sub-banks will be combined with the fast ones when warps contain no more than 16 active threads, therefore, all RF banks become fast.

5.2.4.3 The Implementation of Hybrid RF Bank

To implement the dynamic hybrid RF re-organization under RF-BRO, we switch a certain number of sub-banks from pure slow RF banks with those from pure fast RF banks in the bank organization table during the kernel launch time. For example, sub-bank 3 and 4 are exchanged in Figure 5.6 compared to that in Figure 5.4, and build 2 hybrid RF banks. Two bits are used in each table entry to record every sub-bank's speed information. In a branch induced divergent warp, its register access time on the hybrid RF bank is fast only when all its active threads use the first half registers in a register vector. We compact the active threads of programmatic-value dependent branches into the left-most portion during kernel launch time, so that they are mapped to the fast sub-bank portion of the hybrid RF bank.

During the register access, Eq. 5.1 is applied separately for pure RF and hybrid RF mapping. In the case that there are insufficient hybrid RF banks, some hybrid RF IDs are re-directed to pure RF banks, vice versa. A "read overflow" signal is enabled if the instruction's source pure

RF ID is mapped to a hybrid RF bank at the register read stage. It is also asserted when the active threads in divergent warps fail to compact into the left-most portion. This means that the slow portion of the hybrid RF bank has to be enabled. During a RF bank read, the "read overflow" signal is ANDed with the output of an OR operation on the speed information bits for that bank, determining if a "busy" signal should be generated (shown in Figure 5.6). Moreover, the overflow signal is always asserted when pure slow RF banks are accessed to ensure they have enough time to finish the access operation. Similarly, a "write overflow" signal is used for the write operation.

5.2.5 Putting It All Together

The idea of building hybrid RF banks to hold special register vectors is orthogonal to the warp scheduling policy, we thus propose to combine the two techniques in Section 5.2.3 and 5.2.4 together based on VL-SB and RF-BRO, and maximize both IPC and frequency optimization. Since hybrid RF bank access just takes one cycle in majority of the time, instruction warps contain hybrid RF access is simply treated as fast warp when integrated with the FWAS technique.

The combined technique induces a 16-entry bank organization table, latches, and some simple logic gates to control busy and overflow signals. Both area and power overhead to the sizeable RF is negligible (e.g., less than 2%) based on our gate-level estimation.

5.3 Experimental Methodology

We use a cycle-accurate, open-source, and publicly available simulator GPGPU-Sim (v3.1.0) [19] to evaluate the IPC optimization under our proposed methodologies. Note that our 70% variable-latency technique causes the frequency variations among SMs, and we model that SM level frequency difference into the simulator as well. Our baseline GPU configuration models the Nvidia Fermi style architecture: the GPU contains 15 SMs; the warp size is 32; each SM supports 1536 threads and 8 blocks at most; each SM contains 128KB registers, 16KB L1 data cache, and 48KB shared memory; L2 cache size is 768KB; the scheduler applies the round robin among ready warps scheduling policy. The experimental methodology to model the PV impact and evaluate the frequency optimization under various variable-latency techniques has been described in Section 5.1.1.

We collect a large set of GPGPU workloads from Nvidia CUDA SDK [31], Rodinia Benchmark [32], and Parboil Benchmark [33]. The benchmarks show significant diversity according to their kernel characteristics, divergence characteristics, memory access patterns, and so on.

5.4 Evaluation

In this section, we present the performance results by first evaluating the IPC optimization under various techniques, and then the overall performance improvement which considers the impact on both IPC and frequency. Note that the results of frequency gain under different variable-latency techniques have been discussed in Section 5.1.

5.4.1 IPC Improvement

5.4.1.1 Evaluation of VL-SB with RF-BRO

To evaluate the effectiveness of variable-latency sub-banks (VL-SB) with register file bank reorganization (RF-BRO), we compare it with the VL-RF technique explored by Liang et al. [63]. Figure 5.7 shows the IPC of the investigated benchmarks when those two PV mitigation techniques are enabled. The results are normalized to the baseline case without any optimization (i.e., frequency is determined by the slowest critical path). As discussed in Section 5.2.1, bank0-bank10 are always fast while bank11-bank15 are always slow in all chips under RF-BRO. The IPC results are identical for all chips and only one chip's IPC results for RF-BRO based techniques are shown in Figure 5.7 and 5.8. The averaged IPC results across all 100 chips are shown for 70% VL-RF technique. In the baseline case, all registers have one-cycle access latency, and it has the same IPC as the ideal case without PV. Note that the 70% variable-latency register vector (VL-RV) technique discussed in Section 5.1.2 can also partition the banks into fast and slow sub-banks, thus achieving the same IPC results as VL-SB+RF-BRO. However the frequency improvement of VL-RV is lower than VL-SB+RF-BRO. Therefore, VL-RV is not included in Figure 5.7. As it shows, when compared with the VL-RF mechanism, VL-SB+RF-BRO successfully reduces the IPC loss from 23% to 9% on average across all the benchmarks since it re-organizes sub-banks to deliver a considerable amount of fast RF banks. While VL-RF focuses at a quite fine-grain register level classification, making it impossible to apply the RF-BRO for fast RF bank reorganization and almost all the register vectors accesses take 2 cycles.

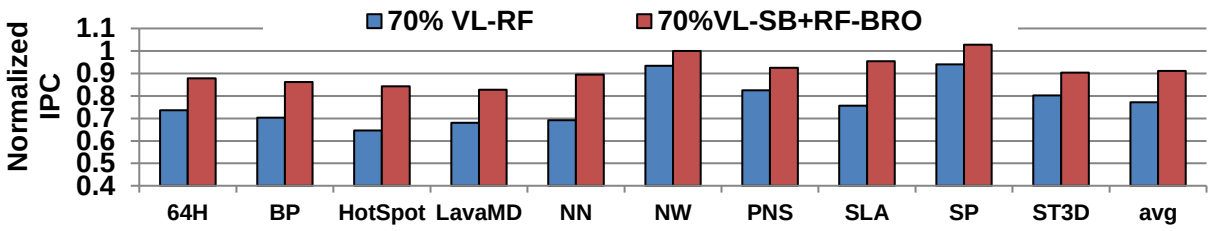

Figure 5.7: Normalized IPC results under 70% VL-RF and 70% VL-SB +RF-BRO.

Interestingly, the IPCs of benchmark NW under both VL-RF and VL-SB+RF-BRO are already approaching to 1. This is because NW includes very few threads, and there are insufficient warps running concurrently in the SM to even hide the stalls for true data dependencies between consecutive instructions from a single warp (in absence of the long memory operations). As a result, the extra register access time is well absorbed by those stall cycles, and has little impact on IPC. On the other hand, the IPC decreases considerably in several computation intensive benchmarks under VL-SB+RF-BRO, because there are few stall cycles helping to hide long RF access delay. For instance, the IPC reduction is 18% in benchmark LavaMD that makes SM active 80% of the total execution time.

5.4.1.2 Evaluation of FWAS

Figure 5.8 presents the normalized IPC results when the fast-warp aware scheduling policy (FWAS) is enabled. We compare FWAS with the fast-bank aware register mapping (named as FBA-RM) discussed in Section 5.2.2. Recently, a two-level scheduling policy has been proposed to boost performance [22]. This policy splits warps into groups and triggers the round-robin warp scheduling policy at intra-group and inter-group level, respectively. By doing this, each group's warps reach a long latency instruction at different points in time which can also alleviate the memory contentions and tolerate the latency. We thus introduce the two-level policy into VL-SB+RF-BRO and compare it with our proposed FWAS policy. As shown in Figure 5.8, all the three techniques are able to mitigate the IPC reduction compared to VL-SB+RF-BRO technique: the IPC losses under FBA-RM, two-level, and FWAS are 7%, 5%, and 1%, respectively.

The effectiveness of FBA-RM is highly related to the register utilization. For instance, NN uses less than 10% of the register file through the entire execution. The fast RF banks are far enough to support the requirement. Moreover, since extremely few registers are utilized

per warp in NN, pushing them to the fast banks negligibility increases the bank conflicts. As a result, the IPC loss of NN decreases from 11% under VL-SB+RF-BRO to only 4% under FBA-RM. Note that VL-SB+RF-BRO applies the default register mapping mechanism, the slow RF banks have the same utilization as the fast ones no matter how many registers are needed, leading to a considerable IPC loss for NN. On the other hand, the performance penalty is severely exacerbated when benchmarks need more register resources. Take HotSpot as an example, it requires around 80% of the register file. When FBA-RM allocates the frequently used register to fast banks, the negative impact caused by the increased bank conflicts outweighs the positive effect of the decreased slow banks accesses, and results in 16% IPC degradation.

As Figure 5.8 shows, the two-level technique integrated with VL-SB+RF-BRO can effectively improve IPC on multiple memory intensive benchmarks, such as 64H and ST3D, as it decreases stall cycles caused by long-latency memory operations. For example, the IPC of ST3D is even 1% higher than that in the baseline case. As an exception, its effect on PNS is quite limited. The group size is fixed (i.e., 8 warps) in two-level policy, and there are only 8 warps (i.e., one group) in most SMs when executing PNS. The inter-group level round-robin in the two-level is inactive.

Similar to the two-level technique, FWAS shows the strong capability to mitigate the IPC loss for memory-intensive benchmarks. Moreover, FWAS does not have any constraint on warp grouping, and it successfully mitigates the IPC loss of PNS to only 2% which is far better than that under the two-level technique. On average across the memory-intensive benchmarks (i.e., 64H, NW, PNS, and ST3D), FWAS boosts the IPC to 105% when normalized to the baseline case, which implies a 12% performance improvement compared to VL-SB+RF-BRO. Additionally, FWAS can effectively optimize IPC for computation-intensive benchmarks, especially those including numerous blocks, because it makes the fast RF banks to support warps at the most degree. For instance, it induces 8% IPC gains for both HotSpot and LavaMD compared to VL-SB+RF-BRO.

5.4.1.3 Evaluation of Hybrid Banks

We further evaluate the normalized IPC results when virtually building hybrid RF banks at kernel launch time for partially active warps, named as hybrid banks. The results of building

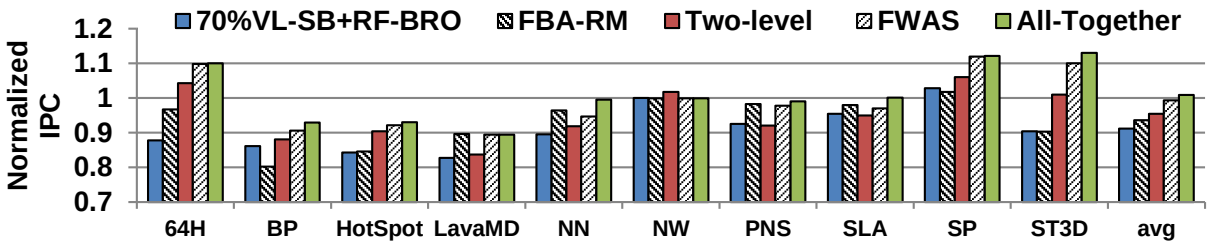

Figure 5.8: Normalized IPC results under FBA-RM, Two-Level, FWAS, and All-Together (70%VL-SB+RF-BRO+FWAS+Hybrid Banks).

hybrid banks on top of all other proposed techniques are presented in Figure 5.8, which is named as All-Together. As expected, the hybrid technique can further improve IPC to even 1% higher than baseline on average across all benchmarks. This is because it directs the special register vectors in hybrid RF banks whose slow portions are rarely utilized. For instance, NN is a typical benchmark that each warp has less than 16 active threads, its IPC result is the same as the baseline case since each RF bank is treated as fast. In addition, all divergent branches in BP are depending on programmatic values, hence, most register vectors in divergent warps are allocated to hybrid banks, which helps to boost the IPC significantly.

5.4.2 Overall Performance Improvement

Figure 5.9 compares the overall performance (IPC×frequency) under the baseline case without any optimization, 70% VL-RF, and our all-together mechanism by presenting the performance distribution over the 100 investigated chips, respectively. The performance is normalized to the ideal case without any PV impacts. As it shows, all-together significantly improves the mean performance by 15% over the baseline case. 70% VL-RF achieves slightly higher frequency (i.e., 2%) than our all-together technique because it performs at fine-grain register level; however, our technique outperforms the 70% VL-RF by 17% due to the substantial (i.e., 24%) IPC improvement.

5.4.3 Sensitivity Analysis

Until now, the random and systematic components have equal variances for both V_{th} and L_{eff} in our PV model (i.e., $\sigma_{rand} : \sigma_{sys} = 1 : 1$). In this subsection, we further evaluate the effectiveness of our proposed techniques when varying the random and systematic component ratios. Figure 5.10(a-b) shows the averaged frequency, and the averaged overall performance

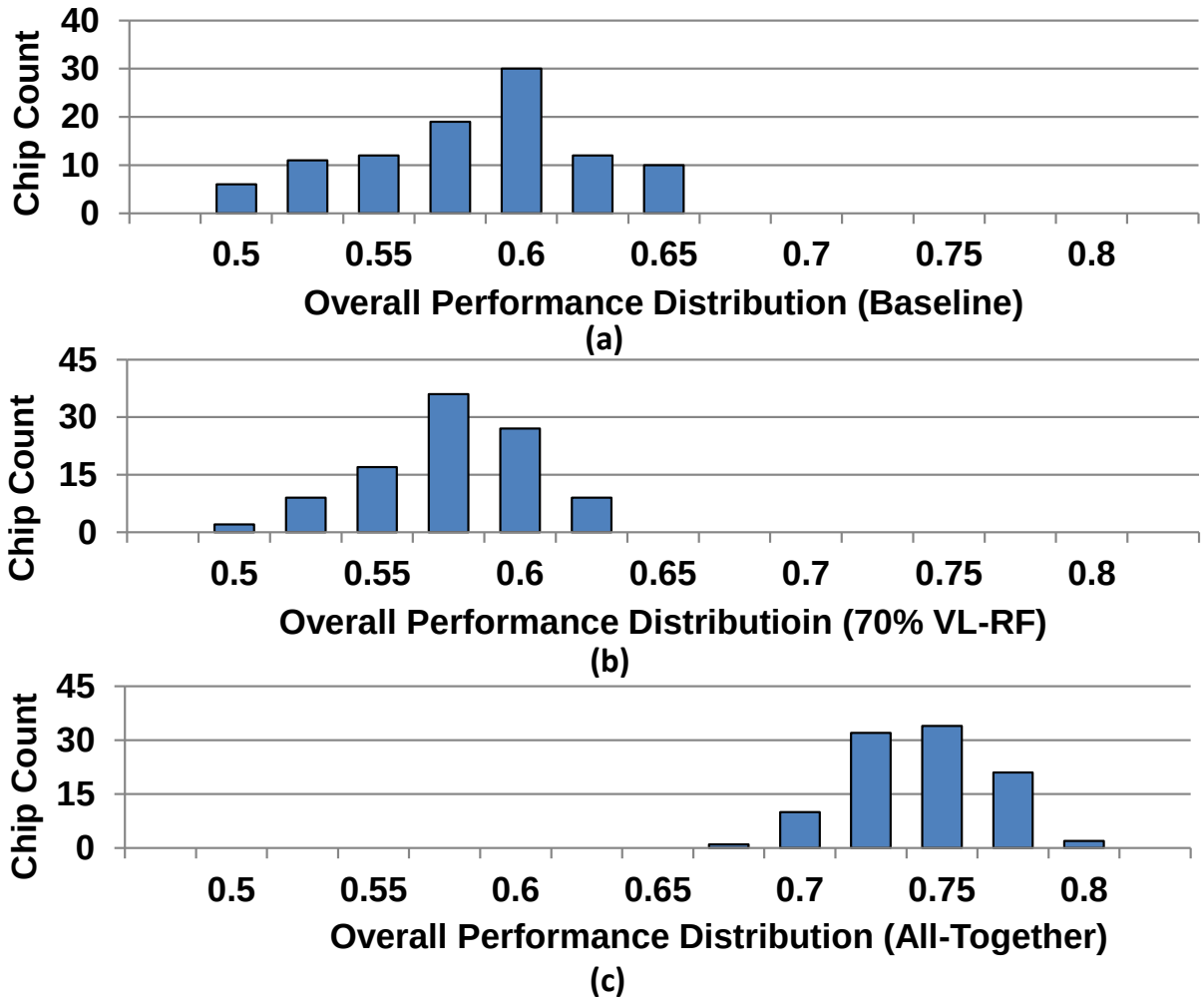

Figure 5.9: Overall performance (IPC×frequency) distribution.

gained by our All-Together technique under various $\sigma_{rand} : \sigma_{sys}$ scenarios. It also shows the standard deviation of the results across the investigated 100 chips. Both the frequency and overall performance are normalized to the ideal case without any PV impacts. As it shows, the frequency of All-Together increases as the systematic variations component increases. Under the scenario of $\sigma_{rand} : \sigma_{sys}$ = 1:4, the frequency loss is only 8% compared to the ideal case without PV impacts. This is because our technique is effective to leverage the systematic variations to provide good frequency boosting. Figure 5.10 also compares All-Together with baseline case without any optimization, 70% VL-RF, and 70% VL-RV. As it shows, the frequency of All-Together is better than 70% VL-RV under all three $\sigma_{rand} : \sigma_{sys}$ scenarios. This confirms that PV exhibits stronger systematic effects in the vertical direction than that in the horizontal direction. Also, the overall performances of All-Together is better than all other three mechanisms under

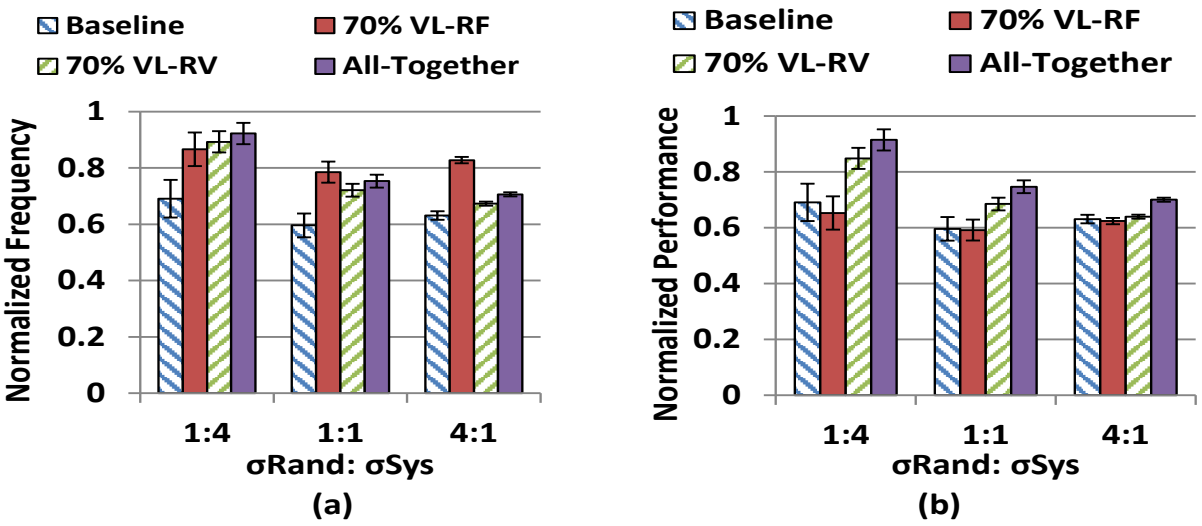

Figure 5.10: Averaged (a) frequency and (b) overall performance (IPC×frequency) results under different random and systematic component ratios. The standard deviations across all chips are also shown in each case.

the three $\sigma_{rand} : \sigma_{sys}$ scenarios. Especially, even when systematic component has minimum impact on variations (i.e., $\sigma_{rand} : \sigma_{sys} = 4{:}1$), it is still 7% better than the baseline case due to its frequency benefit, and 8% better than 70% VL-RF due to its IPC benefit.

Chapter 6

GR-Guard: Enabling Reliable Register File Under Process Variations and Low Supply Voltages

Energy efficiency has become a growing concern for modern GPU architecture design [71, 7, 20, 16, 18, 72, 73, 74, 35, 75]. Current GPUs conservatively operate at high supply voltages to tolerate worst-case scenarios due to static (e.g., process variation) and dynamic (e.g., voltage noise and aging) variations. As a consequence, a large amount of energy is wasted, as power scales quadratically with supply voltage (V_{dd}). Supply voltage reduction at chip level has emerged as a highly effective and promising approach to significantly improve GPU energy efficiency. Recent studies [71, 7] reduce the dynamic portion of the supply voltage guardband by smoothing the voltage noise on GPUs, and the effectiveness of these techniques heavily depend on program characteristics. However, no existing work has tried to lower GPU's V_{dd} below the safe supply voltage limit (V_{min}) by reducing the static portion of the voltage guardband (e.g., to tolerate process variation) determined at the chip-fabrication time, which is program-independent and can be used to further improve the energy efficiency beyond voltage noise smoothing [71, 7] and other traditional energy saving techniques (e.g., dynamic voltage and frequency scaling (DVFS) and power/clock gating). In this chapter, we explore the possibilities of further pushing V_{dd} beyond the V_{min} limit through combating the reliability challenges without hurting the overall performance.

Due to the process variation (PV) effects, the largest on-chip storage structure often determines a chip's V_{min}[76, 77, 78]. PV effects cause variations in transistor parameters due to the imprecise control capability at small feature technology [59, 60, 11, 79]. Such impact from PV

is especially exacerbated on large on-chip storage structures, which usually constitute a large portion of the die area and contain numerous transistors, making the minimum supply voltage required by each cell to function correctly (i.e., $V_{min,cell}$) vary significantly. In order to ensure that all the storage cells operate reliably, the V_{min} is determined by the cell with the highest $V_{min,cell}$. This unnecessarily increases the supply voltage of the other cells and causes energy inefficiency.

Unlike CPUs, the register file in GPUs is usually the largest on-chip data storage structure [51, 56, 80]. In order to support massive thread-level parallelism (TLP) and fast context switch between threads, GPUs are designed to have huge register files to hold the contexts of thousands of active threads [51, 56, 80]. For example, the register file is 3.75MB in NVIDIA Kepler GPUs [56], and the register file size of the recent GPU products is larger than the size of their L1 and L2 cache combined [56, 51, 80]. It occupies a large fraction of the die area and consumes a high amount of chip power [20, 16, 18, 72, 11].

Under a low supply voltage, the GPU register file becomes the reliability hopspot due to its large size and susceptibility to process variations. Moreover, it is frequently accessed by thousands of concurrent threads and is very critical to the GPU performance. Thus techniques that can efficiently handle the faulty register entries at low V_{dd} without hurting the overall performance are highly desired, which may enable substantial energy savings and serve as an essential step to relax the limitation of the minimum V_{dd} requirement for the entire GPU chip. To the best of our knowledge, no such solutions currently exist. Although a variety of circuit- and architecture-level techniques [76, 81, 82, 83, 84, 85, 86] have been previously proposed to tolerate faulty entires at cache and memory level, they are either unable or very costly to be applied to tolerate faulty GPU register entires at low voltage, which will be discussed in detail in a later section.

In this chapter, we address the reliability challenge of the GPU register file at low supply voltage in order to facilitate future aggressive chip-level V_{dd} reduction. First, to understand the reliability issues posed by undervolting, we rigorously model and analyze the process variation impact on the GPU register file at different voltage levels. Based on the model, we observe that only a small percentage of the SRAM cells exhibit large $V_{min,cell}$, which indicates that

substantial energy savings could be gained through proper voltage reduction if these small number of outliers can be tolerated. However, these outliers are spread across the entire register file due to the PV effects, which may translate to a high percentage of the faulty register entries.

To tolerate these faulty register entires at low voltages, we make a key observation by analyzing the modern GPU architectures: due to the massive thread interleaving and the static register resource allocation that naturally exist in GPUs, the time duration that GPU registers contain useless data (i.e., *dead time*) is long, which is even comparable to the time they contain useful data (i.e., *live time*) in the majority of cases. More importantly, there are often many register entries that possess such long dead time at each cycle, and it is rarely affected by different scheduling policies and software-level optimizations. This interesting observation offers great potential to redirect a large number of faulty register accesses to the dead register entries (we refer to these operations as "patching"), granting the capability of aggressive voltage reduction on GPU register file.

Finally, we propose *GR-Guard*, an architectural solution to tolerate faulty GPU register entries at low supply voltage, for achieving significant energy savings without hurting the reliability. GR-Guard enables a simple but effective *Warp Register Group (WRG)* level patching scheme to maintain GPU's efficient SIMT execution pattern, while hiding the patching overhead by leveraging the massive thread-level parallelism in GPUs. It also does not affect the normal (i.e., non-faulty) register accesses. Furthermore, it exploits the unique features of the GPU architecture to minimize its storage, area, and design overhead. For a 28nm baseline GPU, our experimental results show that GR-Guard can enable reliable operations of the register file with less than 2% performance degradation under an aggressive voltage reduction, while achieving an average of 31% energy reduction.

6.1 Reliability Challenge Analysis

In this section, we explore the reliability challenge of supply voltage reduction on the GPU register file under process variation effects through modeling and analysis. The observations and conclusion gained from this process enhance our understanding of the reliability issues and energy-saving potentials of voltage reduction on GPU register file.

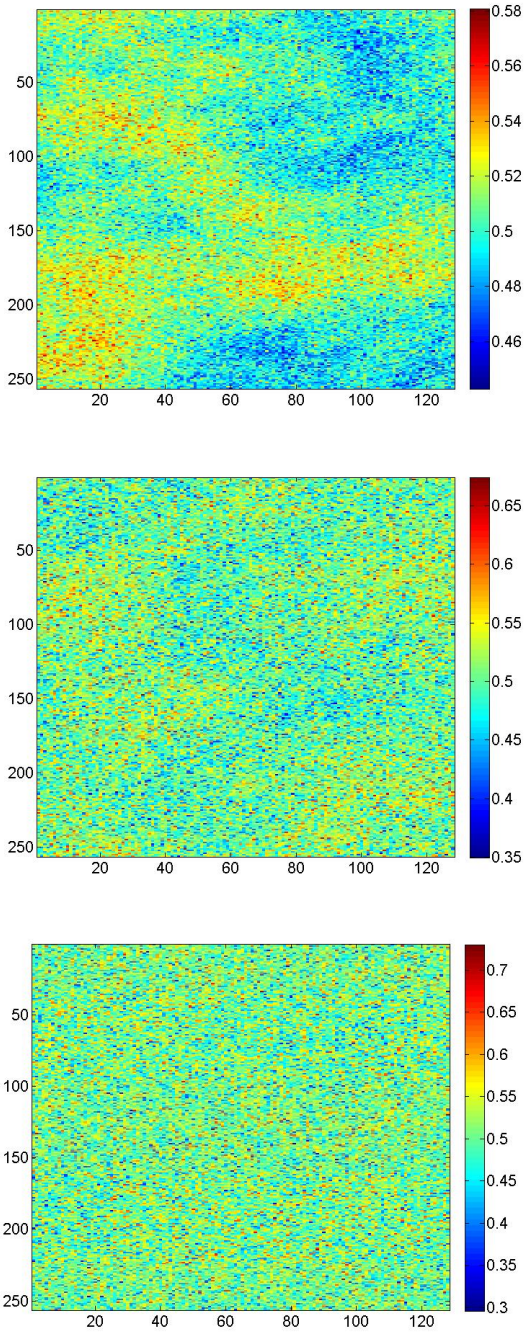

Figure 6.1: $V_{min,cell}$ variation map for a register bank (128 bits $\times$ 256 entries) of 3 different 28nm chips, (left) $\sigma_{rand} : \sigma_{sys} = 1 : 4$; (middle) $\sigma_{rand} : \sigma_{sys} = 1 : 1$; (right) $\sigma_{rand} : \sigma_{sys} = 4 : 1$. Each dot represents $V_{min,cell}$ of each cell.

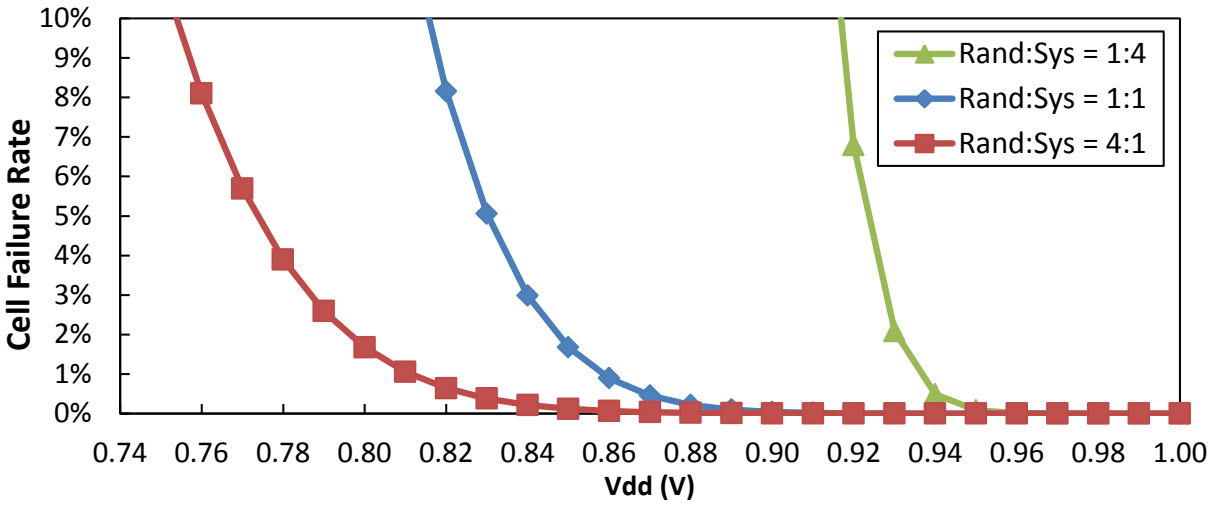

Figure 6.2: The average cell failure rate of the whole register file under different supply voltages (V_{dd}) for 100 chips using 28nm technology.

6.1.1 Modeling Process Variation Effects

Process variation is a combination of random effects (i.e., due to random dopant fluctuations) and systematic effects (i.e., due to lithographic lens aberrations) that occur during manufacturing. Random variations randomly differ the transistor parameters across the chip, while systematic variations cause nearby transistors to share similar parameters. The transistor threshold voltage (V_{th}) and effective channel length (L_{eff}) are the two key parameters that are subject to variations [59, 60, 11]. Generally speaking, the variation of these two parameters can be represented as [59]

$$\Delta V_{th} = \Delta V_{th,rand} + \Delta V_{th,sys} \quad \text{and} \tag{6.1}$$

$$\Delta L_{eff} = \Delta L_{eff,rand} + \Delta L_{eff,sys}. \tag{6.2}$$

To better describe the cell-level reliability impact from voltage reduction, we model each SRAM cell's minimum supply voltage ($V_{min,cell}$) that enables reliable operations. We first model the V_{th} and L_{eff} values of all transistors in the register file under process variation following the VARIUS model [59]. Based on these parameters, we then model the $V_{min,cell}$ value of all the cells by calculating the minimum supply voltage required by each cell to avoid the hold

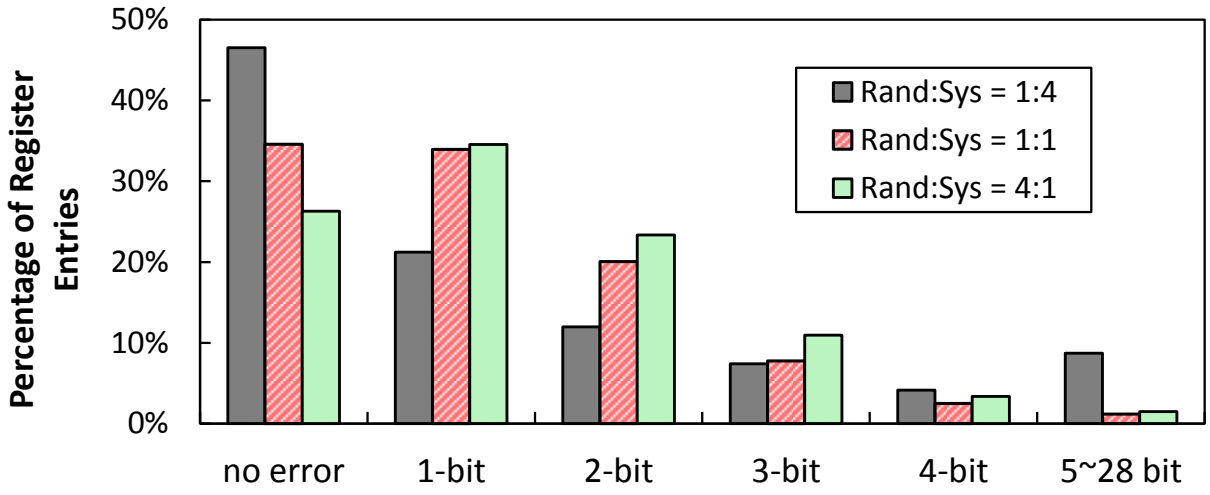

Figure 6.3: Average distribution of register entries with different error bit numbers (i.e., number of faulty cells in a entry) for 100 chips under 1% cell failure rate.

failure and write-stability failure based on the equations used in VARIUS-NTV [60] [1]. Figure 6.1 shows the $V_{min,cell}$ distribution of 3 register banks from 3 chips under different random and systematic impact ratios (i.e., $\sigma_{rand} : \sigma_{sys}$) using 28nm technology (see Section 7.3 for the detailed experimental methodology). As shown, $V_{min,cell}$ exhibits large variations under various random and systematic impacts.

6.1.2 Reliability Challenge of Voltage Reduction

The minimum supply voltage required by the entire register file (referred as $V_{min,RF}$) is determined by the highest $V_{min,cell}$ value among all the cells, in order to ensure all the registers function reliably. If the V_{dd} of the register file is reduced below $V_{min,RF}$, some cells begin to fail. We define a cell as *faulty* if the V_{dd} is lower than its $V_{min,cell}$. We also define the *cell failure rate* as the number of faulty cells within the entire register file under a given V_{dd}. In this subsection, we analyze the impact of supply voltage reduction on cell failure rate.

Figure 6.2 shows the average cell failure rate for 100 28nm chips under three different $\sigma_{rand} : \sigma_{sys}$ ratios. The highest $V_{min,cell}$ in those three chips are normalized to 1V to make the comparison clearer. As shown, under the same cell failure rate, the reduction range of V_{dd} becomes larger as the random impact increases. For instance, under a 1% cell failure rate, V_{dd}

[1] Note that we model the 8T SRAM cell in this work, which is more robust than the traditional 6T SRAM cell under low voltage operations [60, 85]. Read failures do not occur in the 8T SRAM, since the read access uses a different port and cannot flip the cell. Our design proposed in this work though is independent of the SRAM cell structure itself.

can only be reduced by 62mV under $\sigma_{rand} : \sigma_{sys} = 1 : 4$, while it can be reduced by 188mV under $\sigma_{rand} : \sigma_{sys} = 4 : 1$. This shows process variation impact ratio largely affects the value distribution of $V_{min,cell}$. The larger the random impact, the more the $V_{min,cell}$ values are distributed. In contrast, when systematic impact dominates, cells show a more "clustering" effect (Figure 6.1), and tend to have similar $V_{min,cell}$. Figure 6.2 also shows that only a small percentage of the cells have relatively large $V_{min,cell}$ values, called *outliers*. This indicates that significant power reduction can be achieved through proper voltage reduction if these outliers are tolerated.

We further model the location distribution of the outliers by calculating the number of error bits (or faulty cells) within a *register entry* (i.e., consists of four 32-bit registers) under a given cell failure rate. Figure 6.3 shows the distribution of the register entires with different number of faulty bits under 1% cell failure rate. We present the distribution results for three different $\sigma_{rand} : \sigma_{sys}$ scenarios. For space reasons, we group the entries that contain error bits greater than or equal to 5 together. Note that since the register files of recent GPU products [56, 51, 80] are equipped with SECDED (Single-Error Correction, Double-Error Detection) ECC, single-bit errors can be corrected without causing program crash. Thus in this paper, we consider the register entries with no or one faulty bit as *reliable*, and the ones with multi-bit errors, which cannot be protected by SECDED ECC, as *faulty*.

From this figure, we make two observations. (1) Under the influence of process variation impact, even a relatively small percentage of cell failure rate (1%) can result in 31%-39% faulty register entries. This indicates that to gain significant power saving by allowing 1% cells to fail (i.e., 188mV voltage reduction under $\sigma_{rand} : \sigma_{sys} = 4 : 1$ in Figure 6.2), GPUs need to tolerate a large number of faulty register entries (i.e., 31%-39%). (2) When the systematic aspects dominates, the distribution of faulty cells tend to be more clustered, causing the error-bits in some register entries to become very high, e.g., 28-bit errors. We use these observations to direct our proposed design later.

6.2 Fault-Patching Opportunities

To enable a reliable GPU register file under low supply voltage, identifying opportunities for "patching" faulty register entries becomes essential (here "patching" means redirecting the access to one register entry to another). Fortunately, we discovered that the GPU register file

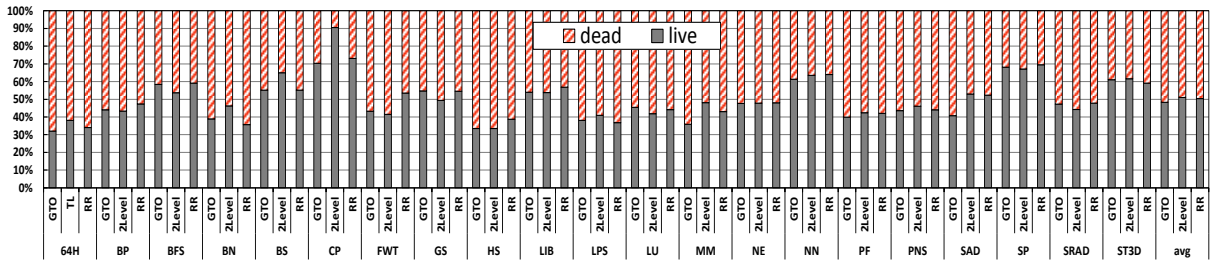

Figure 6.4: Average live and dead time portions for all register values within each benchmark under different warp scheduling policies. GTO: Greedy Then Oldest; 2Level: Two Level, RR: Round Robin.

itself has some unique features that can provide such patching opportunities. First, when the GPU interleaves the execution of thousands of concurrent threads within a simple in-order SM, the time interval between the execution of two consecutive instructions in the same thread is relatively long. Additionally, based on our observation, since each thread has its own register slot in the GPU register file, the access interval to the same physical register is also long. When a register value is dead (i.e., after the value's last read), the physical register contains useless data and will not become live again until the instruction that writes another value to the same register from the same thread is executed. *Live time* represents the cycles between a value's write and its last read. *Dead time* is defined as the cycles between a value's last read and the next value's write.

Figure 6.4 illustrates the average live and dead portions of the residency time (i.e., cycles between a value's write and the next value's write) for all the register values for various representative benchmarks under different warp scheduling policies (see Section 7.3 for detailed experimental methodology). In spite of the differences among the scheduling policies, the average dead time is comparable to the live time for the majority of benchmarks. The only exception is the `CP` application under Two-Level scheduling [20], where the average dead time is only 10% of the residency time. This is because the majority of execution time in `CP` executes a computation intensive loop, which is dominated by back to back instructions. Since the two-level scheduler keeps scheduling the active warps until they are stalled by long latency operations, the register dead time during this loop is greatly reduced. Across all the benchmarks, the average dead time of each register value is between 48% and 51% for different warp schedulers.

To further explore if these dead-time opportunities can be used to patch a large amount of

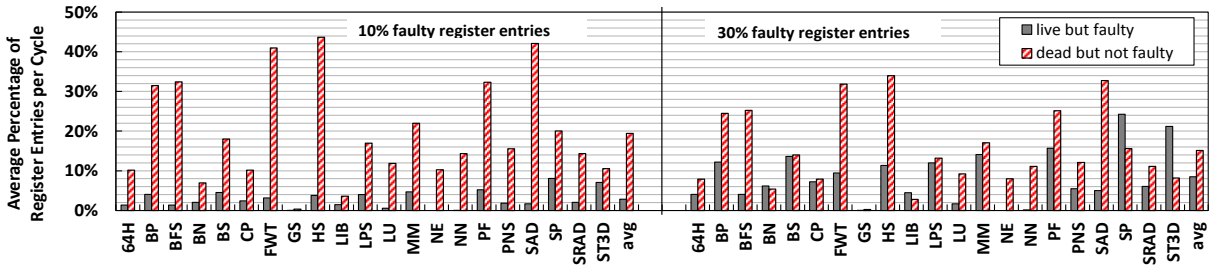

Figure 6.5: Average live register entries that are faulty and dead register entries that are not faulty per cycle for each benchmark, under 10% and 30% faulty rate of register entries using GTO scheduling.

faulty register entries at low voltages, we obtain the average number of live and dead register entries per cycle during the execution of each benchmark. Under a given rate of faulty register entries $N\%$, we then estimate the average number of *live register entries that are faulty* (i.e, number of live register entries $\times$ $N\%$) per cycle, and the average number of *dead register entries that are not faulty* (i.e., number of dead register entries $\times$ (1-$N\%$)) per cycle. Figure 6.5 shows the results under GTO (Greedy-Then-Oldest) scheduling policy [87] for 10% and 30% faulty register entries respectively. Here we only show the results under GTO as an example since the observations for other schedulers are very similar. Across all the benchmarks under 10% faulty rate, approximately 20% of the entries on average can be used to patch (i.e., dead but not faulty) in each cycle, while only 3% of the entries need to be patched (i.e., live but faulty). Similarly, under 30% faulty rate, 15% of the entries on average can be used to patch per cycle, while only 8% of the entries on average require patching. The majority of studied benchmarks have shown that adequate resources from register dead-time are available for fault-patching per cycle, except for `SP` and `ST3D` under 30% faulty rate of register entries. In spite of the exceptions, the register dead-time overall provides promising patching opportunities per cycle, even under a high percentage of faulty register entries. Other possible patching opportunities come from the unallocated registers. In [18], authors observe that a large portion of the register file is not allocated to any thread during program execution for some GPU applications. However, such scenarios are application and optimization dependent. For instance, based on our experiments, the unallocated register portions of some applications such as `SP` and `ST3D` are very small (6% and 3% respectively). Note that all the benchmarks we use are default versions without

aggressive optimizations for higher register usage. Take `SP` as an example, there are only 6% unallocated register entries, but there are on average of 8% and 24% live register entries that require patching *per cycle* under 10% and 30% faulty register entry rate respectively, shown in Figure 6.5. Thus the limited unallocated registers may often be inadequate for patching (see Section 7.4). Additionally, the unallocated entries may also contain multi-bit errors, which further limits this opportunity. Moreover, GPU on-chip resources are designed to support high thread-level parallelism and to be maximally utilized by programmers to achieve the peak performance. The opportunities provided by the unallocated registers heavily rely on applications and optimizations, so they are not suitable to be used as the sole patching resources.

6.3 Fault Occlusion With GR-Guard

In this section, we present our proposed design named *GR-Guard*, which leverages the opportunities that naturally exist in GPUs (i.e., register dead-time) to occlude a high rate of faulty register entries at low voltages. Before elaborating on GR-Guard's design, we want to make clarifications on the following terms. A *faulty entry* is a register entry (128-bit wide) that contains multi-bit errors (Section 6.1.2). A *patching entry* is a *dead but not faulty entry* (Section 6.2) that is used to patch either a faulty entry or another reliable entry that is currently occupied for patching a previous faulty entry (named *occupied entry*). A *warp register group (WRG)* consists of 8 register entries from the same locations of 8 consecutive banks, and contains 32 same-named registers required by a warp. A WRG can include a mix of reliable and faulty entries.

6.3.1 Identifying Register Dead Time

Hardware cannot know if the register's read operation is its last read until another register value is written to the same physical register slot. Therefore, hardware is unable to detect the register dead time in advance. To address this, we leverage the compiler to identify the register dead time opportunities.

We first connect the basic blocks in backward order and perform the live variable analysis [88] at the instruction level. In this way, live register operands can be detected before and after the execution of each instruction. If a register is dead after an instruction execution, this instruction is the register operand's last read and it can be leveraged to patch a faulty entry.

As introduced in Chapter 2, each instruction has at most four input operands based on the NVIDIA PTX standard [16, 11]. To indicate if the corresponding registers of an instruction are dead after its execution, one option is to add one bit for each source operand (four bits per instruction), which could be nontrivial overhead. Since we are only interested in finding the dead registers, the exact live information for all the operands is not necessary. Based on our experiments, most instructions have at most two input operands for almost all the benchmarks. To reduce the overhead without losing patching opportunities, only two bits are added to each instruction to indicate whether the first two input register operands are dead. In this way, the first two input register operands can still be utilized for patching even for instructions with three or four input operands. Note that the compiler itself cannot effectively use the detected dead-register information to patch the faulty register entries, since it treats all the threads the same in GPU. Once a register of a single thread requires patching, the same-named registers in all the threads of the entire program kernel will also be patched, which significantly limits the patching opportunities.

As discussed in Chapter 2, all the active threads within a warp access their same-named registers simultaneously. Thus after obtaining the register operand dead information of each instruction in the compiler, the hardware performs the register dead time identification at the *warp register group* (WRG) level. Control flow divergence may occur during the program execution, which causes different threads within a warp to execute distinct paths. In order to ensure the correctness of register dead time identification (i.e., the active threads in a warp will not mistakenly mark the live registers for the other inactive threads as dead during branch divergence), we only mark the dead registers for non-divergent warp instructions (i.e., all threads in a warp execute the same instruction). Detailed justification for this design choice is shown in Section 6.5.2.

6.3.2 GR-Guard: Structure of the Patching Unit

After obtaining the dead register information, patching can be performed for the faulty entries. We add a *Patching Unit* between the arbitrator and register banks in the GPU register file. Figure 6.6 shows the detailed architecture of the Patching Unit, which includes a *Dead Map*, a *Faulty Map*, an *Index Map*, a *Patch Map*, an *Access Control* logic and a *Patching Selection*

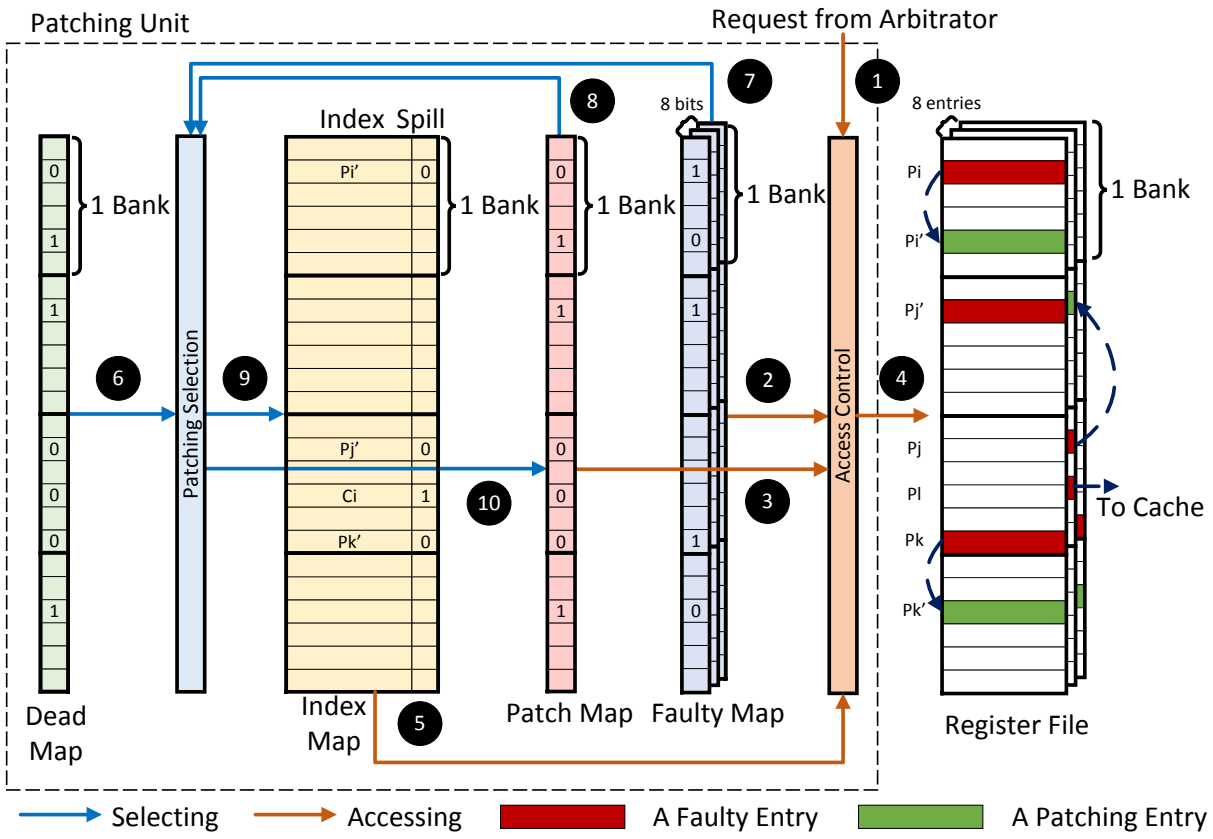

Figure 6.6: The detailed microarchitecture of GR-Guard.

logic.

The *Dead Map* contains one *dead bit* for each *WRG* to indicate if its corresponding register entries are dead. Before kernel execution, all register entries are treated as dead. During the register read stage, the operands' dead information in a non-divergent warp instruction is checked and its physical WRG's dead bit set accordingly. The *Faulty Map* contains one *faulty bit* for each *register entry* to indicate if the corresponding entry is faulty. Whether a register entry is faulty at a given supply voltage can be determined during post-fabrication testing time [11, 85]. The faulty map information can then be set accordingly. Our technique does not change the V_{dd} dynamically, so the faulty map information is fixed for a given chip at runtime. If a different V_{dd} is chosen at chip design time to enable a different cell failure rate, the faulty map changes accordingly.

The *Index Map* contains one entry for each WRG. Each of these entries consists of a *patch index* to indicate the location index of the patching WRG, and a *spill bit* to indicate if the patching is a normal or spill patching (Section 6.3.4)

The *Patch Map* contains one *patched bit* for each WRG to indicate if the corresponding WRG is currently occupied for patching another WRG with faulty entires.

We use the RAM scheme instead of the CAM scheme to implement these four maps in order to reduce the comparison and latency overheads. Additionally, we design these maps to be banked and each bank contains the same number of entries as register file banks, in order to support concurrent requests from different register file banks.

The *Access Control* logic is used to differentiate between the normal and patching access, while the *Patching Selection* logic is used to find dead register entries for patching. Next subsection details the operations of these two logics.

Since the size of the patching unit is very small compared to the register file (Section 7.2.4), the PV impact on it is insignificant. If protection is required, techniques such as redundancy or cell redesign can be adopted to effectively enhance its reliability with negligible overhead.

6.3.3 GR-Guard: Operations

Access Control. Figure 6.6 shows the operations of our proposed GR-Guard. When a register access request is sent from the arbitrator (1), the corresponding *faulty bits* in the Faulty Map (2) and the *patched bit* in the Patch Map (3) are read. Based on this information, the Access Control logic checks if the register entries to be accessed are faulty or occupied entires. If not, these normal accesses are performed the same as in the baseline GPU register file (4). Accessing these two maps is extremely fast due to their very small sizes, and introduces almost no latency overhead to the normal (i.e., non-faulty and non-occupied) register accesses, as we will show in Section 7.2.4. On the other hand, a request to the faulty or occupied entries uses the *patch index* and *spill bit* in the Index Map (5) to obtain the address of the *patching entries* and accesses accordingly (4).

A patching access within the same register bank (e.g., Pi to Pi' in Figure 6.6) can be completed in the following cycle. A patching access to a different bank (e.g., Pj to Pj' and Pk to Pk') requires to send another register access request to the arbitrator, which may encounter a small delay waiting for other pending requests to that bank to be completed. We leverage the operand collectors of the baseline architecture (shown in Figure 2.1) to buffer the pending operand accesses to avoid pipeline stall. Although the execution of a single warp may be slightly

delayed due to patching, this overhead can be well hidden by leveraging the massive thread-level parallelism in the GPU architecture. Note that GR-Guard treats the occupied entries the same as the faulty entires upon accessing, to avoid complicated re-patching. In other words, occupied entries will not be released for their own normal register accesses until completing the current patching.

Patching Selection. When a faulty or occupied entry from a WRG is written, a *patching entry* will be selected. To select an available WRG that contains required patching entries (patching is performed at WRG level, which will be explained later), the Patching Selection logic reads the Dead Map (6), the Faulty Map (7) and the Patch Map (8) to make the decision. GR-Guard gives the patching entries from the same bank higher priority during patching selection. After the decision is made, the corresponding *index* field in the Index Map for the WRG that needs to be patched, will be updated with the location of the selected patching WRG (each WRG has its distinct index) (9). The corresponding *patched bit* in the Patch Map for this selected WRG is also set (10). Once a WRG is dead or written again during execution, its corresponding patching entry or entries (if exists) will be released. During patching, the WRGs for the divergent warp instructions are treated the same as the non-divergent instructions. The complexity of the selection logic is discussed in Section 7.2.4.

6.3.4 Design and Optimizations

WRG-Level Patching. To simplify the design without compromising patching opportunities, GR-Guard performs fault patching at WRG level, instead of at individual register entry level. Only the faulty entries of a WRG are patched, and the faulty entries in one WRG can only be patched to the mapping entries in the other available patching WRG. As shown in Figure 6.6, two entries (the first and the last) from WRG Pk are faulty, and they are patched to their mapping entries (also the first and the last) from WRG Pk'. Note that a WRG that contains faulty entries can also be selected for patching, as long as the required mapping entries are not faulty. For example, in Figure 6.6, even though one entry in Pj' is faulty, its corresponding non-faulty entry can still be used to patch that faulty entry in Pj.

Although patching at register entry level seems to provide the most patching opportunities, it incurs some design issues. First, it breaks the GPU specific feature that register entries within

the same WRG are accessed simultaneously by the same warp, which results in increased design complexity and alters GPUs' SIMT execution pattern. For example, if available register entries within the same physical WRG are used to patch the register entries from different WRGs, this WRG will be prevented from being released for its own normal register accesses. Second, the storage overhead of the current Patch Map and Index Map in Figure 6.6 will be dramatically increased since each register entry will require its own designated slots.

Spill Patching. For the uncommon cases that the patching selection logic cannot find any available WRG for patching, as a backup strategy, the faulty entries will be patched to the L1 data cache as a *spill patching* (e.g., *Pl* in Figure 6.6). Each way in our baseline GPU L1 data cache contains 32 cache lines and the cache line size equals to the size of a WRG (128B). Based on the experiments on a wide range of applications, the spill patching only requires at most 23 cache lines concurrently under a very high percentage of the faulty register entries. Thus we reserve one way in the L1 data cache for spill patching. If there is no spill patching, the L1 data cache functions normally. Otherwise, the cache capacity is slightly reduced by one way during spill patching, and the cache lines in the reserved way are not used for cache accesses. An extra bit is used to indicate if the reserved cache way is used as spill patching, and one bit per cache line in the reserved way is used to indicate if the given cache line is currently used for patching. During spill patching, an available cache line is selected and the address of this cache line is written to the *patch index* of the Index Map (e.g., *Ci* in Figure 6.6). Meanwhile, the corresponding *spill bit* in the Index Map is set to"1" and the selected cache line is marked as used. If all the cache lines used for patching are released, the reserved cache way returns to normal function again. Detailed analysis for the performance and cache miss rate affected by the spill patching can be found in Section 7.4.

6.3.5 Overhead Analysis

Storage Overhead. For analyzing the storage overhead, we recall the structures of the four maps in GR-Guard: the number of entries in the Dead Map, Patch Map, and Index Map is the same as the number of WRGs, while the Faulty Map consists of the same number of entries as the number of register entries in the register file (*RF* for short). For a 128KB RF, Dead Map and Patch Map are both 128B (RF_size/WRG_size = 128KB / (128 × 8)), the size of the

Index Map is 1.375KB ($RF\_size/WRG\_size \times (log_2(num\_of\_WRGs) + 1)$ = 128KB / (128 × 8) × (10 + 1)), while the Faulty Map size is 1KB ($RF\_size/register\_entry\_size$= 128KB/128). For a baseline L1 data cache with 32 sets, only 32 extra bits are required by the reserved way for spill patching. To sum them all together, the total storage overhead is 2.629KB, which is only around 2% of a 128KB register file. We want to emphasize that this storage overhead will remain the same under different cell failure rates and scale well as register file size increases (i.e., almost a fixed percentage of the register file size).

Area Overhead. We use CACTI 6.5 [49] to estimate the area overhead of the added storage structures in the patching unit under 28nm process technology. For the patching selection and access control logic, we apply a similar method used in [89] to aggressively estimate their area overhead as four times of the added storage structures. As a result, area overhead of the patching unit is 0.063 mm^2. For a 28nm-based GPU with 15 SMs, the total area overhead is 0.94 mm^2, which is approximately 0.36% of the GPU chip area.

Access Latencies. We model the access latencies to the Faulty Map and Patch Map using CACTI, since they directly affect the latency of the *normal register access* (Section 6.3.3). Due to their small sizes, accessing these two structures only increases the normal register file access latency by 0.0027ns (for a baseline GPU running at 700 MHz, the cycle time is 1.42ns), which is negligible. Thus we consider the access latencies to the normal registers the same as those in the baseline GPU. We also model the patching selection logic as a tree of 4-input arbiters with priority encoder under 28nm technology in HSPICE. For an input size of 1K, the selection delay is approximately 0.105ns. Also, updating the Index Map and Patch Map only takes 0.018ns. Therefore, we aggressively model the latency of the patching selection as one cycle. Note that if some entries in a WRG require patching, the patching selection process will happen in parallel with the accesses to the normal entries in this WRG.

6.4 Experimental Methodology

Table 7.1 shows the parameters we use to model the $V_{min,cell}$ under the process variation effects for a modern 28nm baseline GPU, as discussed in Section 6.1.1. The basic formulas used in our model are consistent with the previous work [59, 60, 11] and the model is implemented in R [90]. In Section 7.4, we perform sensitivity analysis of our design on three different technology

Table 6.1: Evaluation configurations for the baseline GPU.

Process Variation Parameters	
Sample size	100 chips
Technology node	28nm
V_{thNOM}	0.35V
Correlation Range (ϕ)	0.5
$V_{th}{}'s$ total σ/μ	12%
$L_{eff}{}'s$ total σ/μ	6%
$\sigma_{rand} : \sigma_{sys}$	1:1
GPU Parameters	
Core (SM) frequency	700 MHz
Number of SMs	15
Warp size	32
SIMT pipeline width	32
Warp scheduling policy	Greedy-Then-Oldest (GTO)
Register file size	128KB / SM (32 banks, 256 entries/bank)
Shared memory size	48KB / SM
L1 data cache size	16KB / SM (4-way/128B)
L2 cache size	128KB / memory channel (8-way/128B)
Memory channels	6

nodes, including 11nm, 28nm and 45nm.

We use GPGPU-Sim V3.2.2 [19] to evaluate our proposed technique. The configuration parameters of the baseline GPU architecture are shown in Table 7.1. In order to faithfully evaluate our proposed technique, we collect a large set of representative benchmarks from the NVIDIA CUDA SDK [31], Rodinia benchmark suite [32] and Parboil benchmark suite [33]. We use PTXPlus instruction set simulation in our experiments, which is a one-to-one mapping to the native hardware instruction set, in order to provide a more accurate register lifetime analysis. GPGPU-Sim provides a detailed parser framework for PTXPlus, which includes the control flow graph at the basic block level. We implement our simple register live/dead time detection scheme described in Section 7.2 in GPGPU-Sim.

We use a modified GPUWattch [17] to evaluate the dynamic and leakage energy consumption of GR-Guard under different voltages and frequencies. The frequency scaling factor under V_{dd} reduction is modeled by calculating the frequency of a 50-stage FO4 inverter chain [65] under 28nm technology and different V_{dd} values in HSPICE. Additionally, the energy consumption of the patching unit and cache accesses due to spill patching are also included in our evaluation. The V_{dd} of the baseline GPU chip is 1V, which includes the voltage guardband to tolerate the process, voltage, and thermal variations and aging. In order to solely evaluate the effectiveness

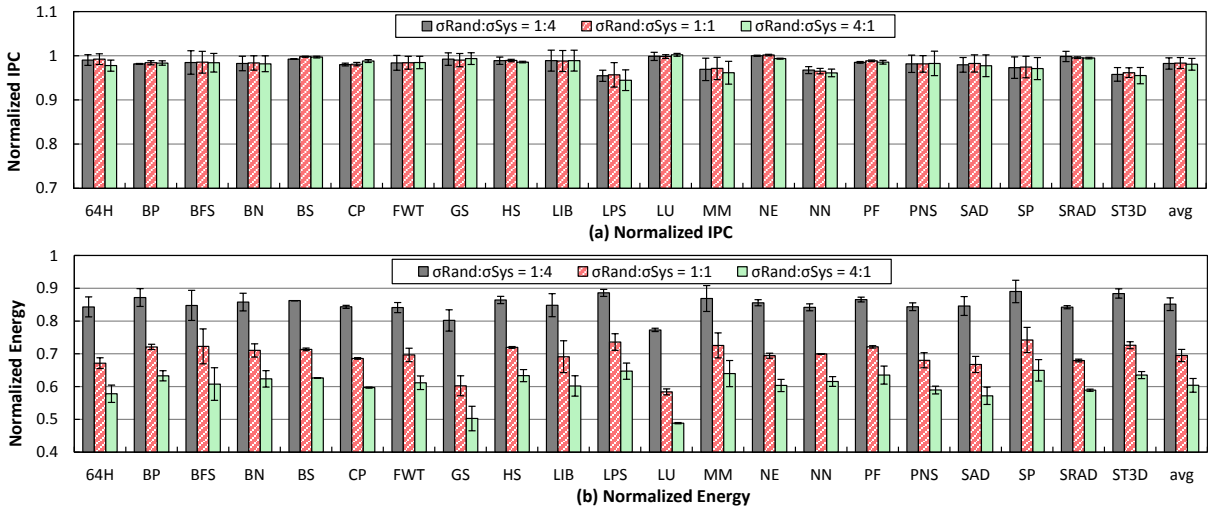

Figure 6.7: The average (a) IPC and (b) energy of the GPU register file with GR-Guard under a 1% register cell failure rate and different $\sigma_{rand} : \sigma_{sys}$ ratios across 100 chips.

of GR-Guard, we assume the worst case scenarios under other source of dynamic variations (e.g., voltage droop, thermal effects) in all the experiments. Also, only the GPU register file is evaluated with the reduced supply voltage in our evaluation.

To better evaluate the effectiveness of GR-Guard, we simulate a cell failure rate of 1%, which translates to a high percentage of faulty register entries (between 31% to 39%) on a 28nm baseline GPU (Figure 6.2). We believe this is a reasonable value to achieve the desired yield for manufacturing.

6.5 Experimental Results

This section details experimental results and analysis for our proposed design. First, we explore the overall performance and energy impact of GR-Guard. Then, we quantify different patching opportunities through analyzing the breakdown of register accesses. After that, we explore if the spill patching in GR-Guard affects the L1 data cache performance. Finally, we conduct several sensitivity studies, including process technologies, scheduling policies and energy scalability, to explore various design choices for GR-Guard. Without specific mention, *all the results are averaged across 100 chips under 1% cell failure rate and 28nm technology.* The results are also normalized to the baseline scenario without register file voltage reduction.

6.5.1 Performance and Energy Impact

Overall Performance (IPC). Figure 6.7(a) shows the average IPC of the GPU with GR-Guard enabled under different $\sigma_{rand} : \sigma_{sys}$ ratios. The +/-3 standard deviations (3σ) are also shown in the figure, which comprises 99.7% of all values. Figure 6.7(a) shows that overall GR-Guard achieves good performance across all benchmarks under 1% cell failure rate and three process variation ratios. On average, it only incurs negligible IPC degradation (less than 2%) for all the three scenarios. Several factors contribute to this low performance overhead of GR-Guard when combating a high percentage of faulty register entries: (1) the patching opportunities provided by the register dead time are quite adequate; (2) the high TLP of GPU effectively hides the patching overhead during execution; and (3) the WRG-level patching mechanism in GR-Guard maintains GPU's highly-parallel SIMT execution pattern. Also, on average, performance is slightly worse under a larger random impact (σ_{rand}). This is because although the register cell failure rate (1%) is consistent among all three scenarios, the number of faulty register entries could be slightly larger under a higher random impact due to a less clustered faulty cell distribution.

Additionally, under a higher random impact ($\sigma_{rand} : \sigma_{sys} = 4 : 1$), computation intensive benchmarks, including `LPS` and `MM`, have experienced a small IPC degradation under GR-Guard: 5% and 4% respectively. This small performance degradation is due to the overhead incurred from patching a single warp. Although such overhead can normally be hidden quite well in GR-Guard, it becomes more visible for computation intensive cases due to their high pipeline occupancy. The other case with a small IPC loss (4%) is `NN`. Since most of its kernels contain only one warp, there is not enough warp-level parallelism to hide the patching overhead. Additionally, since `SP` and `ST3D` show less adequate patching opportunities under high percentage of faulty entries (Section6.2), GR-Guard incurs 3% and 4% IPC degradation for them respectively due to spill patching, which will be analyzed in detail in the next subsection.

Energy Reduction for GPU Register File. Figure 6.7(b) shows the average register file energy with GR-Guard enabled under different $\sigma_{rand} : \sigma_{sys}$ ratios. As shown, GR-Guard can effectively tolerate a high failure rate of register entries caused by a large voltage reduction, resulting in substantial energy savings. On average, with the support of GR-Guard, the energy

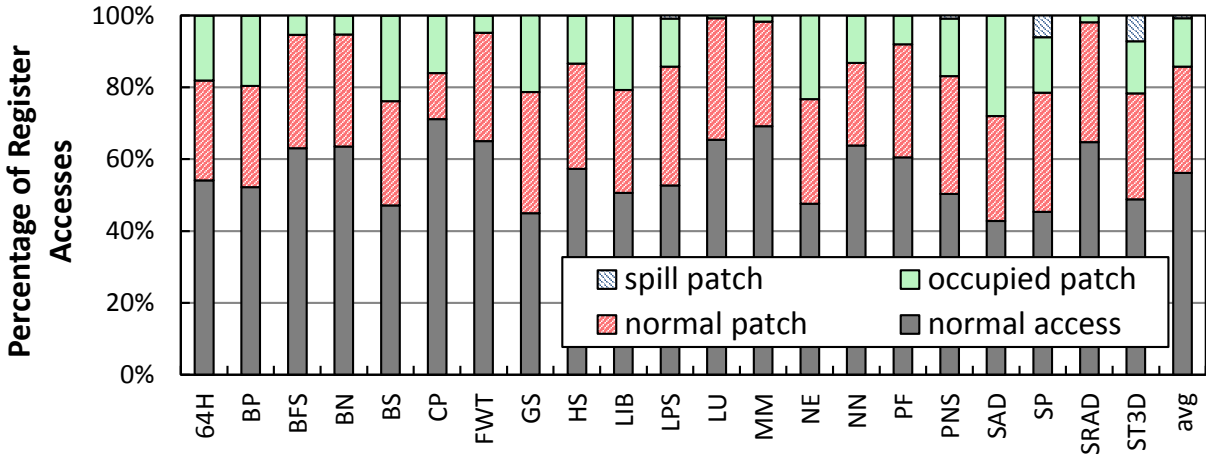

Figure 6.8: The breakdown of the register accesses under GR-Guard for a 1% register cell failure rate.

consumption of GPU register file is reduced by 15% under $\sigma_{rand} : \sigma_{sys} = 1 : 4$, 31% under $\sigma_{rand} : \sigma_{sys} = 1 : 1$, and 40% under $\sigma_{rand} : \sigma_{sys} = 4 : 1$, compared to the baseline case. The figure also shows that a larger random impact can result in higher energy savings when undervolting. This is because $V_{min,cell}$ has a larger variation under a higher random impact, which results in a bigger V_{dd} *reduction range* compared to the other cases that are under the same cell failure rate (e.g., Figure 6.2 in Section 6.1.2).

From this point on, we focus on a more balanced impact ratio of $\sigma_{rand} : \sigma_{sys} = 1 : 1$ in the following evaluations.

6.5.2 Evaluation of the Patching Opportunities

Figure 6.8 shows the breakdown of the register accesses under GR-Guard for a 1% cell failure rate. On average, 56% of the register accesses under GR-Guard are *normal accesses*, which means that GR-Guard does not incur overhead for more than half of the register accesses. 30% of accesses are *normal patching accesses*, which are used to patch most of the faulty register entries. There are also 13% of the accesses used for patching the occupied entries, which indicates that the dead time of the occupied entries is not long enough for the requested patching duration so GR-Guard has to patch the normal accesses to these entries somewhere else if they are not released on time. Finally, on average, only around 1% of the accesses are for *spill patching*. This indicates that the register dead-time opportunities are enough to patch almost all the faulty register entries in most benchmarks under a high failure rate. Among 21 applications,

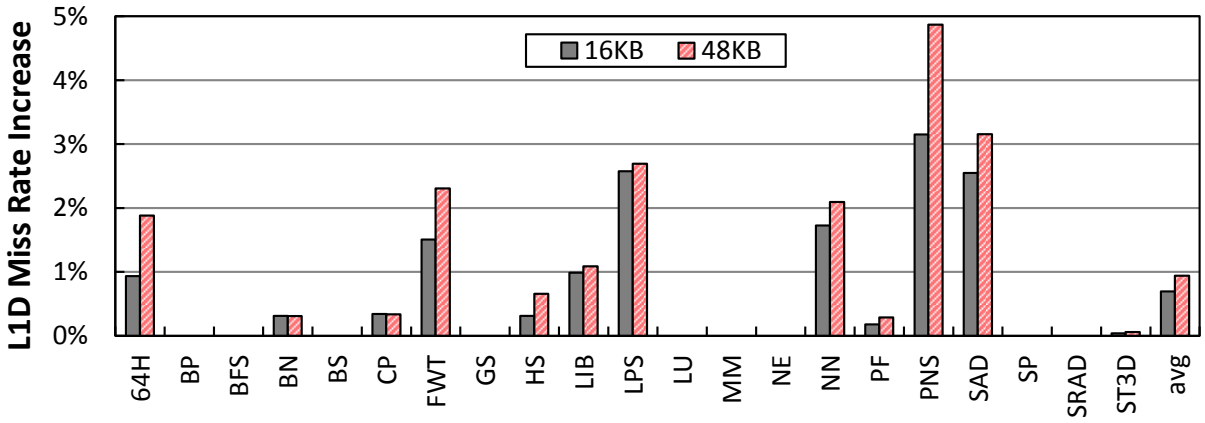

Figure 6.9: The increase of L1 data cache miss rate under GR-Guard for a 1% register cell failure rate.

only `SP` and `ST3D` show a small percentage of register accesses for spill patching (6% and 7% respectively) due to the large number of live registers. However, compared to the other accesses, this number is still small and has trivial impact on the application performance, demonstrated in Figure 6.7(a).

Recall Section 6.3.1, for simplifying the design and ensuring the correctness of register dead-time identification, at hardware-level GR-Guard excludes the divergent warp instructions from the selection pool. Figure 6.7(a) shows that for benchmarks with high divergence, including `BFS`, `GS`, `HS`, `LPS`, `LU`, and `SAD`, no significant performance degradation is observed. Additionally, Figure 6.8 demonstrates that no significant spill patching occurs for these benchmarks either. Take `HS` as an example, although 46% of warp instructions are divergent, there is no spill patching in the register accesses. This proves that the register dead-time opportunities in the non-divergent warp instructions are adequate for patching a high percentage of faulty register entries at the low voltage, without the need of further increasing the design complexity and overhead to identify the additional opportunities from the divergent warp instructions.

6.5.3 The Impact on Cache Miss Rate

Discussed in Section 6.3.4, as a backup strategy, GR-Guard reserves cache lines in one way of the L1 data cache (L1D) for spill patching when it cannot gain any patching opportunity from the dead registers. Here we evaluate the impact of GR-Guard design on the miss rate of the baseline 16KB and a larger 48KB L1D. For a fair comparison, we keep the number of

sets the same for the two configurations, while only increasing the number of ways by 3 times in the 48KB L1D. Figure 6.9 shows that the average L1D miss rate across all applications only increases by less than 1% for both configurations. The L1D miss rate increase is slightly higher for the 48KB configuration, since the reduction in miss rate from a larger cache size slightly raises the impact of GR-Guard on the cache performance. However, GR-Guard does not noticeably affect the performance for both configurations due to the small difference in the L1D miss rates. As a relatively cache-friendly benchmark, `PNS` has the highest cache miss rate increase, which is only around 3% in 16KB L1D and 5% in 48KB L1D, respectively. The minor increase in cache misses can be attributed to the small amount of spill patches shown in Figure 6.8. Also, in GR-Guard, the L1D capacity is only reduced by one way during spill patching, so for the majority of the time, L1D is used as a normal cache. As shown in Figure 6.8, despite `SP` and `ST3D` having a relatively larger amount of spill patches, they incur almost no increase in cache miss rates. This is because both `SP` and `ST3D` exhibit little data locality and almost all the cache accesses are misses. On the contrary, `BFS` and `LIB` are cache sensitive and have relatively good data locality. However, they have no and only 1% cache miss rate increase under GR-Guard, respectively. This is because they have no or very small amount of spill patches. So for the L1D miss rate to increase significantly under GR-Guard, applications need to have both good data locality and a large amount of spill patches, which is quite uncommon.

6.5.4 Sensitivity Analysis

Scheduling Policies. Figure 6.10(a) shows the impact of different warp scheduling policies (i.e., GTO, 2Level, and RR) on register file IPC, Energy and EDP (energy-delay product) through GR-Guard. The results are averaged across all the benchmarks for 100 chips. Although different warp scheduling policies slightly change the average live/dead time portions of register values, their overall impact on performance and energy is almost the same.

Process Technologies. Figure 6.10(b) shows the effectiveness of GR-Guard under 1% register cell failure rate and three different process technologies (i.e., 45nm, 28nm, and 11nm). Under the same cell failure rate, the IPC degradation across different process technologies is very similar, which is around 2%. Since the process variation impact becomes larger under smaller feature sizes, GR-Guard achieves the highest energy savings for the register file under

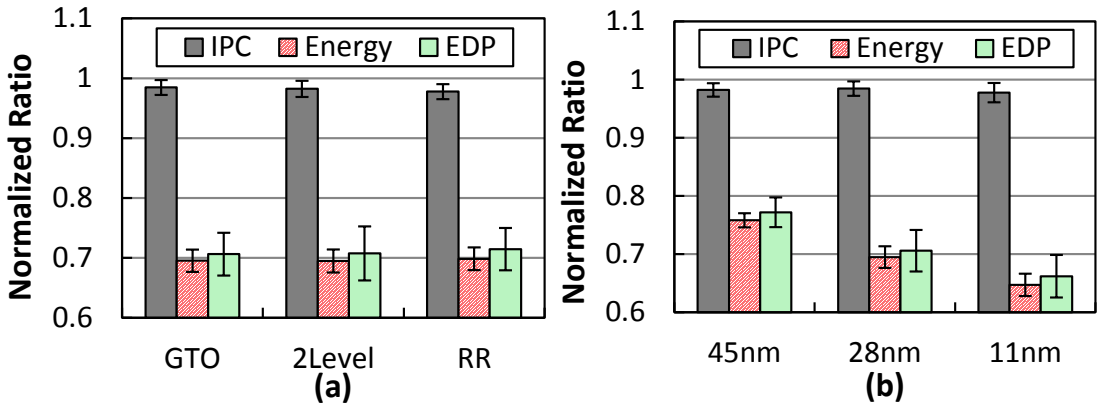

Figure 6.10: The normalized IPC, energy, and EDP of GPU register file with GR-Guard enabled under (a) GTO, 2Level, and RR scheduling policies, and (b) 45nm, 28nm, and 11nm. The ±3 standard deviations are also shown in the figure.

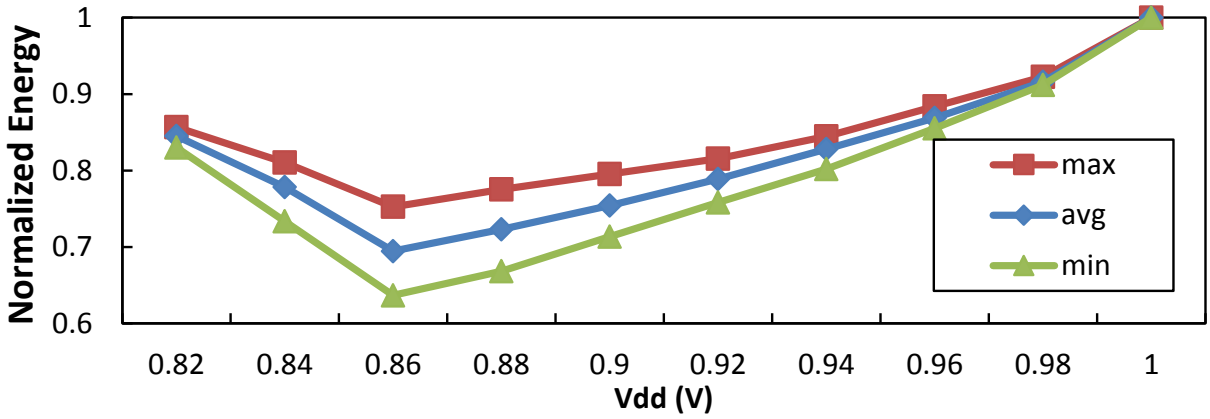

Figure 6.11: Energy scaling curves under V_{dd} scaling.

11nm due to its larger supply voltage (V_{dd}) reduction range, while still maintaining reliability. On average, the baseline energy consumption of GPU register file is reduced by 24%, 31%, and 35% under 45nm, 28nm, and 11nm, respectively, through GR-Guard.

Energy Scalability under Voltage Reduction. Figure 6.11 shows the energy scaling curves of the GPU register file under voltage reduction. The average, maximum, and minimum results are selected across all the benchmarks for 100 chips under 28nm technology. This figure helps us identify the optimal scaling point for register file V_{dd} in order to achieve the maximum energy savings. Since the cell failure rate increases as V_{dd} is reduced, we can observe that at 0.86V ($\sim$ 1% cell failure rate) GPU register file has the minimal energy consumption through GR-Guard. Further reducing V_{dd} beyond this point will cause the extra power savings from the lower voltages to be offset by the overhead of tolerating the high percentage of faulty register entries (>40%).

Chapter 7

LoSCache: Leveraging Locality Similarity to Build Energy-Efficient GPU L2 Cache

With the unprecedented high computational throughput, modern GPUs are extensively applied for accelerating high performance computing (HPC) applications. To improve performance, GPUs exploit high thread-level parallelism (TLP) to hide the long latency memory accesses. On-chip caches are also adopted to enhance the data locality of GPU applications. However, recent studies [12, 13, 91, 92, 93, 94, 95, 96] have suggested that the cache hierarchy is not efficiently utilized in current GPUs. They have been primarily focused on addressing the inefficiency of L1 cache (e.g., intelligent cache bypassing) due to its performance-criticality. In contrast, as a much larger and lower memory hierarchy, efficiency of the L2 cache is largely ignored in the previous work because it is not the primary performance bottleneck [93, 94]. However, as a large on-chip SRAM structure [17], inefficient utilization of L2 cache can significantly increase the overall GPU energy consumption.

To better understand the L2 cache efficiency, we first analyze its utilization. Figure 7.1 shows an example of the request sequence for a cache line. In this example, the cache line starts to store useless data after the last request of a certain data. We refer to the time periods that a cache line stores useless data as "dead time", labeled in Figure 7.1. As it shows, storing these useless data unnecessarily increases energy consumption. Based on our experiments on a wide range of applications, we observe that L2 cache lines spend an average of 95.6% of the time being dead. This indicates that the current GPU L2 cache is poorly utilized and consumes a significant portion of energy in storing data that will not be re-referenced again in future.

To further identify the root cause of this phenomenon, we then characterize and analyze the data locality of GPU applications. We find that the majority of data sharing happens at

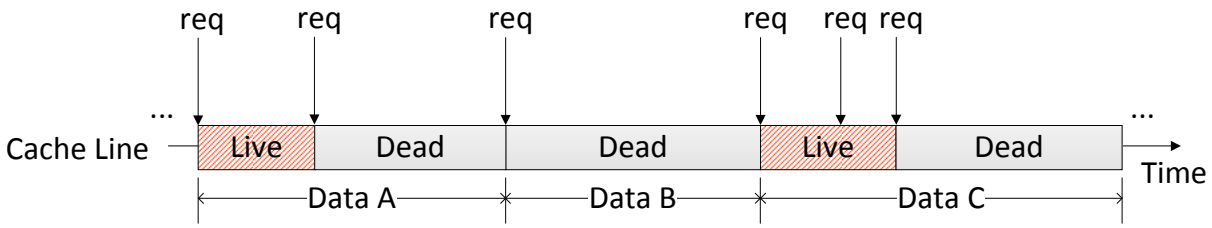

Figure 7.1: Illustration for GPU L2 dead time. "Dead time" represents the time periods of storing useless data.

intra-CTA (i.e., cooperative thread arrays) level in GPU. As the L2 cache is shared by different CTAs, the lacking of inter-CTA locality in applications leaves the L2 cache unsuitable for most of the data accesses. By further analyzing the data re-reference counts for all the L2 accesses, we observe that the majority of the L2 data has no or few future re-references. These findings suggest the inefficient utilization of L2 is caused by the data locality characteristics of the current GPU applications.

To reduce the unnecessary energy waste caused by L2 underutilization, we need to identity and then maximally reduce these dead-time periods. Finally, we make a key observation for this paper: the data locality has instruction-level similarity among different threads due to the unique feature of the SIMT programming model of GPU. This finding can be used to accurately predict the data re-reference counts at L2 cache block level. With this information, L2 cache lines can be powered off during the "dead time". Based on this idea, we propose a simple energy-efficient L2 design named *LoSCache* to reduce the unnecessary energy waste in L2 caused by its utilization inefficiency. LoSCache performs locality prediction at data level instead of cache line level. It uses the locality information from a small group of CTAs to accurately predict the L2-level data re-reference counts of the remaining CTAs based on the locality similarity. With that information, LoSCache can then power off cache lines if they are predicted to be dead after certain accesses to conserve energy. Experimental results show that LoSCache can reduce the L2 cache energy by an average of 64% with only 0.5% performance degradation due to its high prediction accuracy. Our sensitivity studies also demonstrate that the effectiveness of LoSCache is independent of different L1 cache designs and scheduling policies.

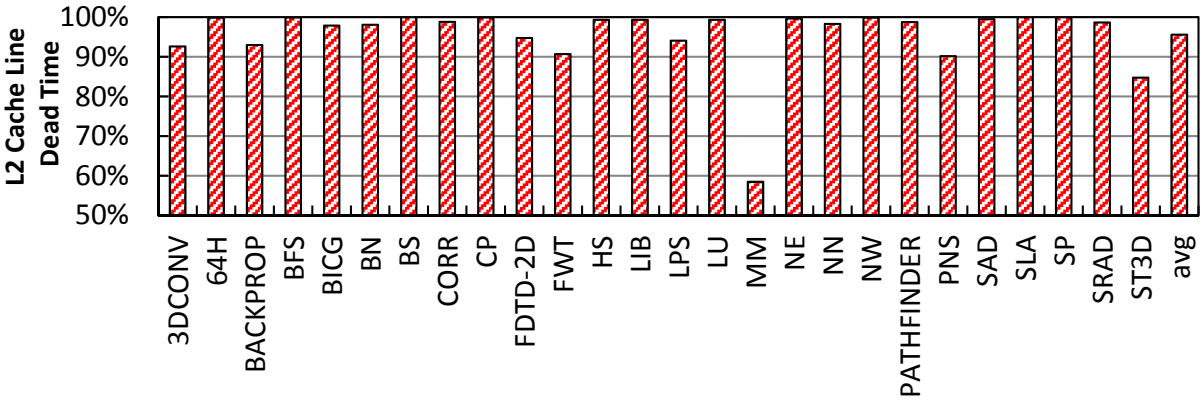

Figure 7.2: L2 cache line dead time on baseline architecture.

7.1 Motivation And Findings

In this section, we first illustrate the utilization inefficiency of GPU L2 cache through profiling the L2 cache line dead-time for a wide range of applications. To further investigate what causes such long dead-time for the L2 cache lines, we then characterize and analyze the GPU data locality in terms of intra-CTA and inter-CTA. After that, we look into the distribution of the data re-reference and identify the root cause for GPU L2 inefficiency: only a small percentage of data has inter-CTA locality with some re-references. Finally, we show our discovery for *instruction-level data locality similarity* on GPU, which can be used to predict the data re-reference counts in L2 and eventually help address the energy waste issue from the long dead-time in L2 cache lines.

7.1.1 L2 Utilization Inefficiency

Recall the illustration of dead time in GPU L2 in Figure 7.1, we want to investigate how much time the L2 cache lines spend on storing useless data. Figure 7.2 illustrates the dead-time percentage of L2 cache lines across a wide range of benchmarks [1]. The percentage value is calculated as $Total\_Dead\_Time/(Execution\_Time \times Num\_Cache\_Lines)$. The figure clearly shows that on average L2 cache lines spend 95.6% time on storing data that will not be referenced again, indicating the inefficient utilization of GPU L2. Among all the benchmarks, only `MM` has relatively lower (i.e., $< 60\%$) cache line dead-time. To find out the root cause of this utilization inefficiency, we study the GPU data locality patterns next.

[1] The detailed experimental methodology on the baseline architecture and benchmarks is introduced in Section 7.3.

7.1.2 Intra-CTA and Inter-CTA Locality

As illustrated in Figure 2.1, since L1 cache is private to each SM and L2 cache is shared by all the SMs, they have different data preferences. In other words, data should be placed to its appropriate cache hierarchy to achieve the maximum performance benefit. Since the thread allocation, communication and synchronization all occur at the CTA level, we classify the locality of memory data into intra-CTA and inter-CTA locality. Figure 7.3 shows two examples for these two locality cases. *Intra-CTA locality* (Figure 7.3(a)) describes the data that is only accessed by threads within the same CTA, which includes both intra-warp locality (i.e., the data is only accessed by threads from the same warp) and inter-warp intra-CTA locality (i.e., the data is accessed by threads from different warps within the same CTA). *Inter-CTA locality* (Figure 7.3(b)) describes the scenario that data is accessed by threads from multiple CTAs. As threads within the same CTA execute on the same SM, ideally, the per-SM L1 data cache is suitable to store the data with intra-CTA locality and these data should bypass the L2 cache to leave the capacity to the data with inter-CTA locality.

In order to study the GPU data locality characteristics, we first partition the GPU memory space into the same-size chunks, and the size of each chunk is as same as the cache line size (e.g., 128B). We then check if each data chunk is only accessed by memory requests from a single CTA or from multiple CTAs in each benchmark. The data chunks that are only accessed by threads from a single CTA are considered to have intra-CTA locality, while the data chunks that are accessed by threads from multiple CTAs are classified as having inter-CTA locality. Note that inter-CTA locality category has two subcategories, including inter-CTA locality on the same SM and across SMs. Figure 7.4 shows the percentage of data chunks with intra-CTA and inter-CTA locality in each benchmark. As shown in this figure, the majority of memory data chucks have intra-CTA locality, which accounts for 69% on average across all the benchmarks. In contrast, on average only 31% memory data chunks have inter-CTA locality, which indicates that GPU applications do not tend to share data among CTAs, which is consistent with the GPU programming model. Figure 7.4 also shows most benchmarks (18 out of 26) are dominated by intra-CTA locality, except for `3DCONV`, `HS`, `LPS`, `MM`, `NN`, `SLA`, `SRAD`, `ST3D`, in which the majority of data exhibits inter-CTA locality. For these applications that are dominated by inter-CTA

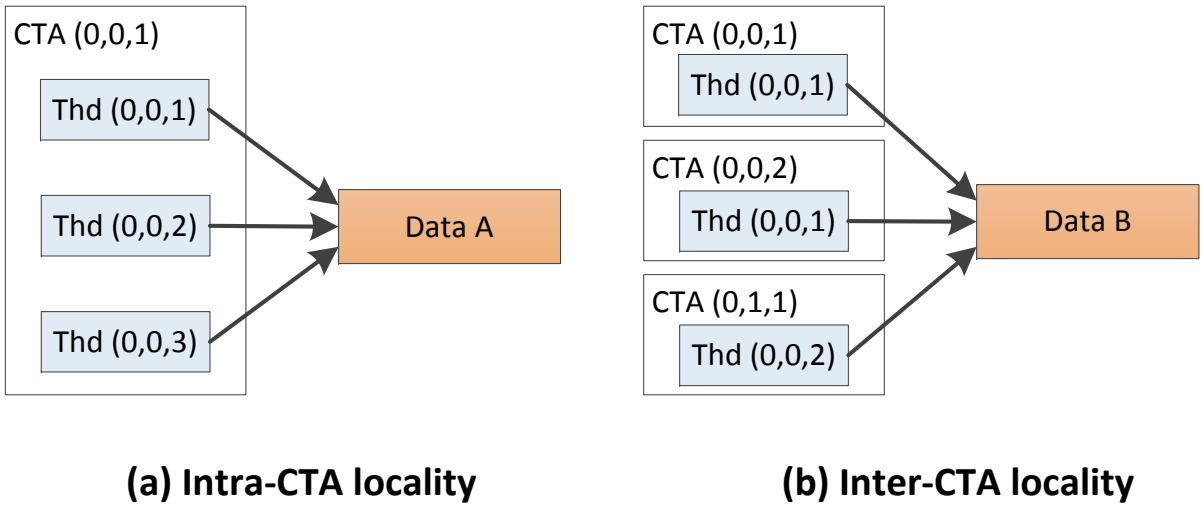

Figure 7.3: Examples of (a) intra-CTA and (b) inter-CTA data locality, respectively.

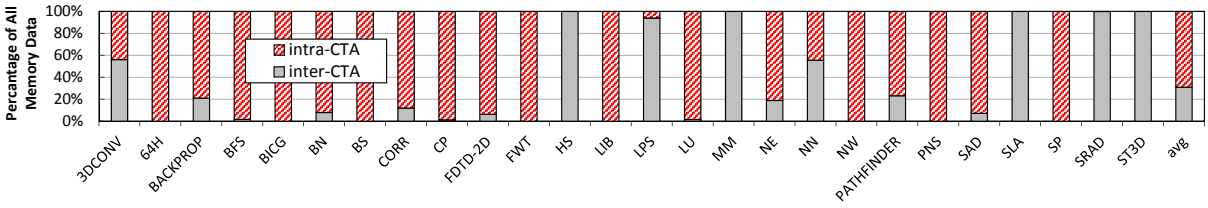

Figure 7.4: Percentage of memory data (128B) with intra-CTA locality and inter-CTA, respectively.

locality, only `3DCONV` has less than 1% of the data that has inter-CTA locality on the same SM due to the common GPU CTA scheduling pattern. Other applications do not contain data with inter-CTA locality on the same SM. Thus, we only consider the data accessed by threads from multiple CTAs on different SMs to have inter-CTA locality in this paper.

7.1.3 Root Cause of L2 Utilization Inefficiency

To explore the root cause behind L2 utilization inefficiency shown in Figure 7.2, we analyze the *data re-reference distribution* in L2 cache in this subsection. The counting for data re-reference is conducted at the L2 cache line level, instead of evaluating the individual memory access. We calculate the cache block address by masking the cache block offset of each memory access address. More specifically, we treat all accesses to the same L2 cache line address as accessing the same data, no matter if the actual data sizes of the requests are different or if the data has been evicted before re-referenced.

We then calculate the re-reference counts for the L2 data accesses in all the benchmarks and

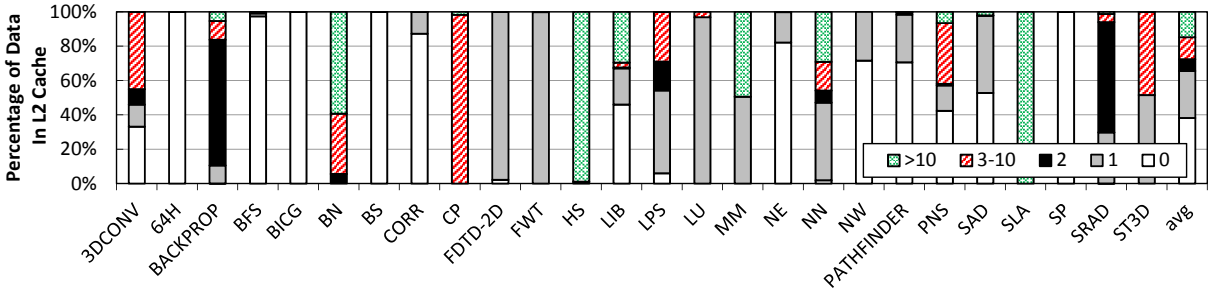

Figure 7.5: Data re-reference counts in L2 cache.

the results are shown in Figure 7.5. From Figure 7.5, we observe that most data accessed in L2 cache has low re-reference count due to the lack of inter-CTA locality. On average across all the benchmarks, 38% of the data will not be re-referenced again by future L2 cache accesses, which indicates that these data either has intra-CTA locality that has been captured by the L1, or has no locality due to streaming (e.g., `64H`, `BICG`, `BS`, and `SP`). Additionally, 27% of the data on average is re-referenced only once. Only 34% of the data in L2 cache is re-referenced more than once, the majority of which has inter-CTA locality. The rest of it has intra-CTA locality but is referenced in L2. This is because data is written once or multiple times before it is read by the same CTA, or thrashing occurs in L1. As the L1 data cache is write-no-allocate in GPU, the data write will not be cached in L1 until the same data is read in the future. The above observations from Figure 7.5 reveal the root cause of the L2 utilization inefficiency: majority of the stored data in L2 has no or only one future re-reference, and even the data with inter-CTA locality still has relatively low future re-reference counts. This motivates us to design a more energy-efficient L2 cache: if we can predict the data re-reference counts in advance, we can apply power-gating to shut off some cache lines during the time intervals in which they store useless data (or dead-time shown in Figure 7.1).

7.1.4 Instruction-Level Data Locality Similarity

In order to know the data re-reference counts, we need a mechanism to accurately predict it in advance. Fortunately, we discover that GPU applications have an unique instruction-level data locality similarity behavior that can be used to predict data locality. In this subsection, we introduce this observation. The exact prediction technique for data re-reference will be discussed in Section 7.2.

GPU uses a SIMT programming model, which means all the threads within the same kernel

execute the same program code with different operands. Thus when instructions with the same program counter (PC) are executed by different threads, they tend to exhibit similar behaviors. As a result, the data access requests generated by the same-PC instructions from different threads tend to have similar locality behavior as well. We explain the reason by analyzing the memory access instructions from two representative benchmarks, `LIB` and `MM`. From Figure 7.4, we can observe that data accesses of `LIB` are dominated by intra-CTA locality, while `MM` is dominated by the accesses with inter-CTA locality. Figure 7.6 and 7.7 further illustrate two code examples for data accesses from these two benchmarks. For `LIB` shown in Figure 7.6, every thread loads data from the global memory to its own shared memory space (line 9) based on the CTA ID and thread ID (line 3). Since the CTA ID and thread ID are both one dimensional in `LIB`, the data accesses have intra-CTA locality. Thus, even though different data is accessed by different threads when executing the same-PC instructions in this code, the data tends to show similar locality behavior. Figure 7.8(a) further explains this locality similarity. Three threads from CTA (0,0,1) all access data A1 when executing instruction i with PC 1, so data A1 has intra-CTA locality. When the other three threads from CTA (0,0,2) execute the same instruction i with PC 1, they all access Data A2. Although Data A1 and A2 are different data, they have the same locality behavior. Note that locality similarity we discuss here is the number of re-references to the accessed data. For instance, Data A1 and A2 are both re-referenced twice. We call this characterization as *instruction-level data locality similarity*.

Figure 7.7 shows another code example from `MM`. In this case, every thread loads the data from the global memory (i.e., A and B) to their own shared memory space (i.e., AS and BS) (line 13 and 14) only based on the thread ID and one dimension of the block id (i.e., blockIdx.x or blockIdx.y) (line 8 and 9). When executing the same instruction, threads from different CTAs access the same data from the global memory as long as their thread IDs and one dimension of their CTA ids are the same, resulting in the exhibition of inter-CTA locality. In addition, the re-reference time in L2 for shared data is related to the number of CTAs that have the same dimension value. *"Related" but not "equal" is because there are cases where CTAs with the same dimension value locate on the same SM, which belongs to intra-CTA locality.* Figure 7.8(b) further illustrates this case. Thread (1,1,0) from CTA (2,0,0) and thread (1,1,0) from

```
1   __global__ void Pathcalc_Portfolio_KernelGPU2(float *d_v)
2   {
3     const int    tid = blockDim.x * blockIdx.x + threadIdx.x;
4     const int threadN = blockDim.x * gridDim.x;
5     int   i, path;
6     float L[NN], z[NN];
7     for(path = tid; path < NPATH; path += threadN){
8       …
9       d_v[path] = portfolio(L);
10      …
11    }
12  }
```

Figure 7.6: Code example of `LIB` with intra-CTA locality.

CTA (3,0,0) load the same data "B1" from the global memory when executing inst j with PC 2. Thus "B1" has inter-CTA locality. Similarly, when thread (2,2,0) from CTA (1,3,0) and thread (2,2,0) from CTA (2,3,0) execute the same instruction j, they both access data "B2" . Although "B1" and "B2" are different data, they show the same inter-CTA locality characteristic as they are generated by the same instruction j. For example, both Data B1 and B2 have 1 re-reference at L2 cache, if both B1 and B2 are not accessed by other instructions and all the CTAs with the same value in the block ID are allocated to different SMs. Otherwise, the re-reference counts to B1 and B2 are different but still highly similar. As shown in these examples, data locality in GPU applications has unique instruction-level similarity for different threads. We will leverage this opportunity to predict the data re-reference count next.

7.2 Energy-Efficient L2 Cache Design Using Data Locality Similarity

In this section, we propose a mechanism that leverages the instruction-level data locality similarity among SIMT threads to build an energy-efficient L2 cache. We first describe how to use the data locality similarity to accurately predict data re-reference counts. We then demonstrate how to leverage this technique to design an energy-efficient GPU L2 cache, named *LoSCache*. Finally, we provide the optimization details and overhead analysis of LoSCache. It

```
1   __global__ void matrixMul( float* C, float* A, float* B, int wA, int wB)
2   {
3       int bx = blockIdx.x;
4       int by = blockIdx.y;
5       int tx = threadIdx.x;
6       int ty = threadIdx.y;
7       …
8       int aBegin = wA * BLOCK_SIZE * by;
9       int bBegin = BLOCK_SIZE * bx;
10      …
11      for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
12          …
13          AS(ty, tx) = A[a + wA * ty + tx];
14          BS(ty, tx) = B[b + wB * ty + tx];
15          …
16      }
17      …
18  }
```

Figure 7.7: Code example of `MM` with inter-CTA locality.

is worth noting that the design goal of LoSCache is to complement the current state-of-the-art L1 cache designs [12, 13, 91, 92, 93, 94] to achieve significant additional energy savings from GPU L2 cache without affecting the performance. The core of the LoSCache design is to gain the maximum energy savings from the "dead time" of L2 cache lines illustrated in Figure 7.1.

7.2.1 LoSCache: Structure

In order to implement LoSCache, we add minor hardware extensions to the LD/ST unit and L2 cache of the baseline GPU architecture, shown as the red-highlighted components in Figure 7.9. We discuss the details of these two hardware extensions as follows.

Prediction Table: Shown in Figure 7.9, a prediction table is placed in the LD/ST unit, which is aimed to predict the re-reference counts of data at L2 cache line level (Section 7.1.3) based on the instruction-level data locality similarity discussed in Section 7.1.4. Some control logic are also added together with the prediction table. Each entry in the prediction table contains the locality prediction information about L2 cache accesses from instructions with an

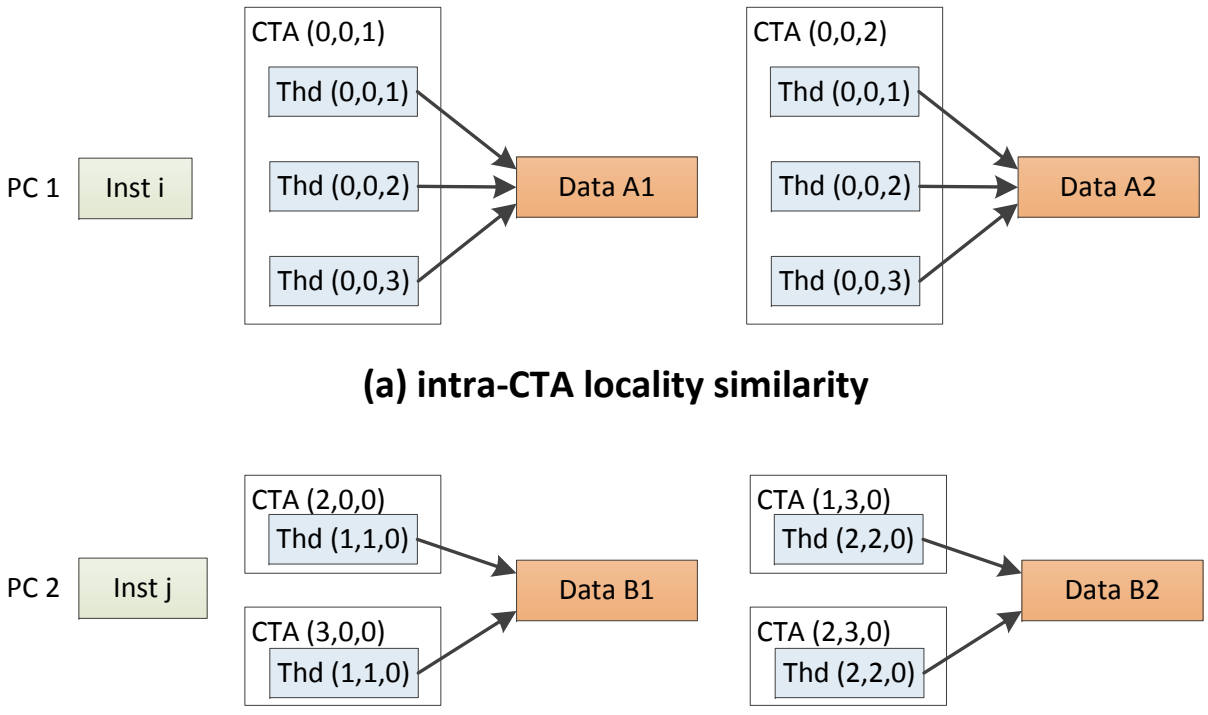

Figure 7.8: Examples for instruction-level (a) intra-CTA and (b) inter-CTA locality similarities.

unique PC. Figure 7.9(a) shows the details of a prediction table entry, which contains the *block address* of the L2 cache line, the *PC* of the instructions that access this L2 cache line, and the predicted *access count* to this L2 cache line. The *PC* and the *access count* are used to indicate the predicted locality while the *block address* is used to locate the corresponding L2 cache line at the end of the prediction phase (Section 7.2.2).

L2 Cache Extension: In order to assist the locality detection, we also extend the tag part of the L2 cache, as Figure 7.9 shows. Each cache line is extended with an *access count* field to indicate the actual access count of the current data in this cache line. When the current data is evicted, the access count field is reset to 0. This field is used to update the locality information of the prediction table at the end of the prediction phase, and find the dead time of the corresponding cache line for energy savings during the normal execution. The size of the hardware extensions and their overheads (e.g., area and latency) will be analyzed in Section 7.2.4.

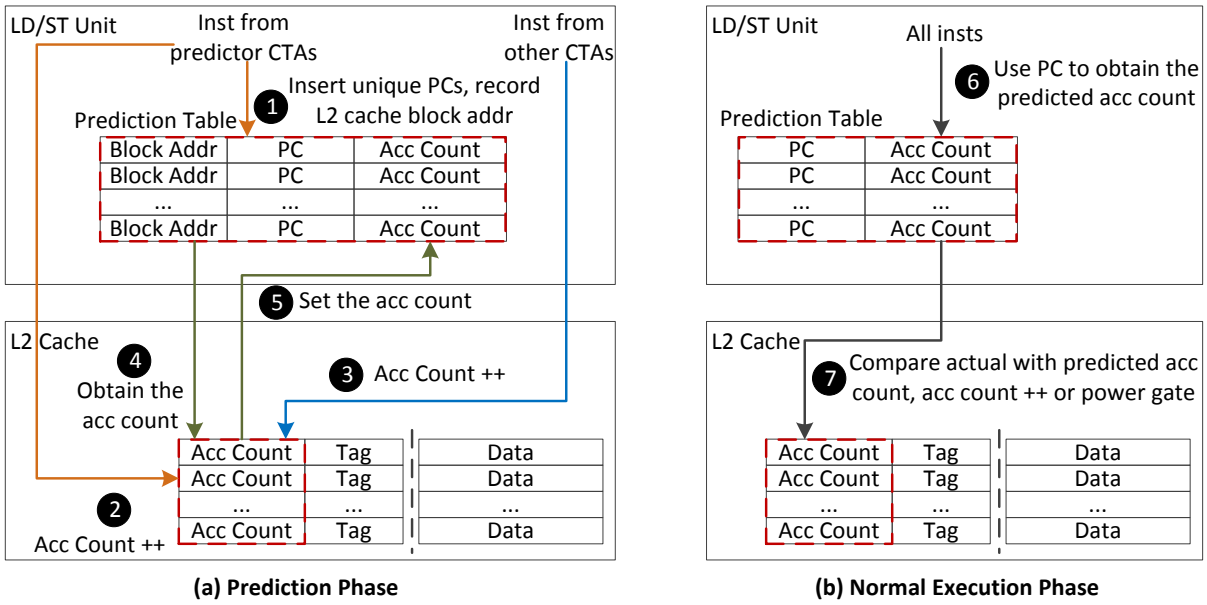

Figure 7.9: Hardware extensions and implementation of LoSCache.

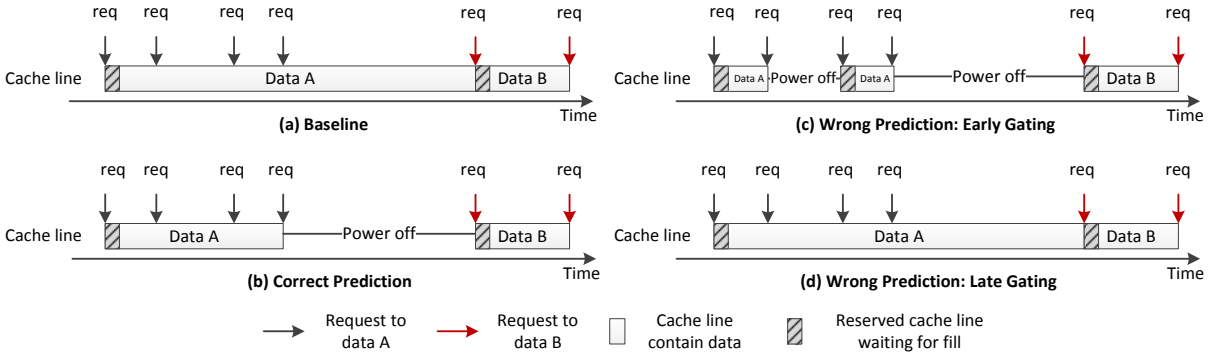

Figure 7.10: Prediction scenarios, including Correct Prediction, Early Gating and Late Gating.

7.2.2 LoSCache: Operation

In this subsection, we introduce how to leverage the instruction-level data locality similarity to design energy-efficient GPU L2 cache. We discuss the three major operation phases in details as follows:

Predictor Selection. During the program execution, different threads tend to execute in a similar pace on GPU because of the programming model and hardware scheduling. In addition, the access interval to the same memory address is usually long due to the massive thread interleaving. Thus it is very difficult to predict the data locality of each memory request under the current scheduling policies (e.g., Greedy-Then-Oldest [12]). Our basic idea to address this issue is to select some "predictors" and execute them faster in order to make runtime decisions

for the others. A simple way to implement this on GPU is to detect the locality information of all memory accesses in a single thread, and use it to predict the locality for all the other threads within the same kernel. However, an accurate prediction will not be reached until the predicting thread completes a large amount of the memory access instructions. Also, the faster execution of this thread will be impeded by the CTA-level barriers. Moreover, the locality information of a single thread may be biased. In this work, we propose an efficient mechanism to cut down the drawbacks from single thread prediction. We randomly select one CTA from each SM as the *predictor*. This is because the characteristics of certain CTAs may be dramatically different from the others for the same application. For example, in one kernel of `BFS` [97], certain threads from a CTA are assigned with more workloads than the others, which means these threads have more memory accesses. So using the requests from this CTA to predict for the other CTAs will be inaccurate. Our random predictor selection reduces the possibility of selecting the abnormal CTAs as the predictors. Additionally, choosing multiple CTAs across SMs as predictors reduces the bias.

Prediction Phase. After the predictors are selected, the program first goes through the *prediction phase*. During this phase, threads within the predictors (i.e., CTAs) have higher execution priority. Meanwhile, threads from other CTAs execute in a lower priority, which means their ready threads will not execute until none of the threads in the predictors are ready (e.g., stalls due to data dependencies). In this way, the predictors can generate the locality information in advance for prediction (see Section 7.2.3 for optimization on prediction period). Figure 7.9(a) shows the prediction phase. When memory access instructions from the predictor CTAs are issued, their PCs and the L2 cache block addresses are inserted into the prediction table in parallel with the memory access request generation (1). The prediction table of each SM only stores entries with unique PCs. So before a PC is inserted into a table, it will be first compared with PCs of the existing entries in the same table. If there is a match, this PC will not be inserted. Meanwhile, the memory access requests are generated in LD/ST units and sent to the L2 cache. The corresponding access count of the L2 cache line is increased by one from both the predictor CTAs (2) and the other normal CTAs (3). At the end of the prediction phase, the access count information of each entry in the prediction table is updated: using the

block address of each entry to locate the corresponding L2 cache line (4), and then update the access count of the prediction table using the access count of the corresponding L2 cache line (5).

Normal Execution. Once the prediction phase is completed, the information in the prediction table becomes available for use. We refer to this phase as the normal execution phase, shown in Figure 7.9(b). During this phase, all the memory request instructions check the prediction table using their instruction PCs to obtain the predicted access count of the corresponding L2 cache line (6). This is in parallel with the address calculation in the LD/ST unit. When accessing to the targeted data, the predicted access count is then compared with the current access count of the L2 cache line (7). If the current access count is smaller than the prediction, the current access count is increased by one. Otherwise, it means the access is the last one according to the prediction. Thus the corresponding L2 cache line is powered off after this access in order to save energy. If the predicted access count is 1, the accessed data will not be stored in the L2 cache because it is predicted not to be referenced in L2. Note that all the L2 cache lines are powered off at the beginning of the program and wake up when first accessed for extra energy savings. This is because some applications are observed to only use a small percentage of L2 cache lines during the entire execution.

Figure 7.10 illustrates several prediction scenarios. For the baseline case (Figure 7.10(a)), six memory requests are sent to the same L2 cache line. The first four access data A, and the last two access data B. As this figure shows, the cache line contains useless data during the time between the last access to data A and the first access to data B. Under correct prediction (Figure 7.10(b)), the cache line is powered off during this time period to save energy without hurting the performance. If the predicted re-reference count is smaller than the actual re-reference count to the cache line for the entire program, the cache line is powered off earlier (i.e., early gating). As Figure 7.10(c) shows, it is powered off after the second request to data A. So the third request becomes a miss and the cache line is woke up again, hurting the performance. Figure 7.10(d) shows the case of late gating, which means the predicted re-reference count is larger than the actual re-reference count of the data. In this case, the cache line will not be powered off during the time it contains useless data. As a consequence, there will be no energy savings (i.e., same

as the baseline case in Figure 7.10(a)).

7.2.3 Optimizations

Setting Prediction Period. The prediction period is critical to the accuracy and effectiveness of the prediction. The most naive way is to terminate the prediction period when all the predictor CTAs finish their execution. However, for applications that have a large amount of L2 accesses, this greatly reduces the effectiveness of the prediction since a decent prediction could very likely be available earlier than that time, which limits the energy-saving potential. Thus we also consider the L2 cache access intensity when the prediction period is determined. Based on the experiments, we set 100 L2 cache accesses as the threshold. So the prediction period is finished after 100 L2 cache accesses or all the predictor CTAs are completed, whichever comes first.

Increasing Prediction Accuracy. We apply an adaptive mechanism to increase the prediction accuracy for the early gating scenario (Figure 7.10(c)). After a cache line is early power-gated, the tag still stores the information before the next data fills in. If the same data hits the tag of this cache line, it indicates that the actual re-reference count of the data is larger than the predicted. We then add a threshold for the future prediction of the accesses from this PC. In this case, the re-reference count for the future memory accesses is equal to the predicted value plus the threshold. Every inaccurate prediction will increase the threshold by one. Based on our experiments, a maximum threshold of three covers almost all the applications that we study. Therefore, we set its upper bound to be three.

7.2.4 Overhead Analysis

Prediction Table Size. The prediction table size is related to the number of unique PCs for the L2 cache accesses during the prediction phase. Figure 7.11 shows the maximum number of unique PCs for each application. Based on the figure, `NW` has the largest number of unique PCs, which is 21. So we use it to set the number of entries for the prediction table. For each table entry, we set 25 bits for the L2 block address, 32 bits for the PC field, and 6 bits for the predicted access count (i.e., the highest L2 access count among all applications is 32). Therefore, we estimate the size of the prediction table as 21*(25+32+6)=165 Bytes per LD/ST unit.

L2 Tag Extensions. We set the access count in the L2 cache line to 6 bits, same as that

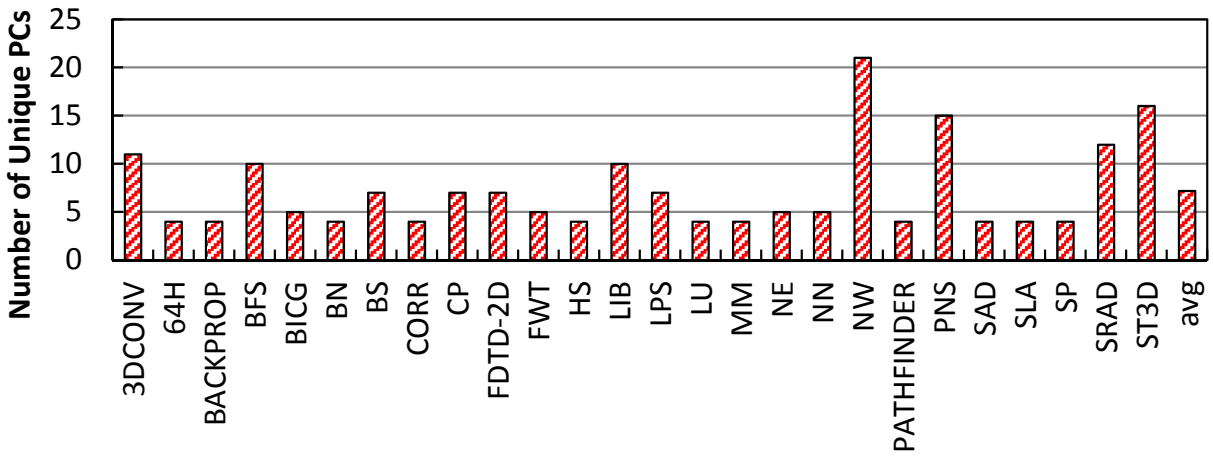

Figure 7.11: Maximum number of unique PCs in the prediction table.

in the prediction table. Thus the storage overhead is 6 bits per L2 cache line.

Area Overhead. We use CACTI 6.5 [49] to calculate the area overhead of the added storage structures under 40nm technology. For the control and gating logic, we apply a similar method used in [89] to aggressively estimate their area overhead as three times of the added storage structures. Based on our evaluation, the area overhead is $0.449mm^2$, which is only 0.084% of the entire GPU chip area.

Latency Overhead. We assume accessing the prediction table takes 1 additional cycle. Even though it cause additional latency, it is well hidden since this process can be done in parallel with the memory address generation in the LD/ST units. Additionally, there are wake-up periods for the power-gated cache lines. However, the wake-up process does not negatively affect the performance because this process is in parallel with the data access to the lower level memory hierarchy. In other words, the cache-line reservation periods require to fetch new data from the memory during cache misses anyway. Finally, at the end of the prediction phase, setting the access count in the prediction table causes more interconnect network traffic. We will analyze it in Section 7.4.2.

7.3 Experimental Methodology

Simulation Environment: Our proposed LoSCache is evaluated using GPGPU-Sim V3.2.2 [19], which is a widely-adopted cycle-accurate simulator for GPU architecture research. The configuration parameters of the baseline GPU architecture are shown in Table 7.1. The design choices applied to our proposed design is discussed in Section 7.2.4.

Table 7.1: Baseline Architecture Configuration

Parameters	Value
SIMT Core (SM)	15 cores, SIMD width=32, 1.4GHz, 5-stage pipeline
Max/SM	1536 threads, 32768 registers, 48 warps, 32 MSHRs with 256 entries
L1 Cache/SM	16KB/core, 128B line, 4-way assoc, 1-cycle hit latency
Shared Mem/SM	48 KB; 32 banks; 3-cycle latency; 1 access per cycle
Unified L2 Cache	768 KB, 128KB/bank, 6 banks, 128B line, 16-way assoc
DRAM	6 memory channels, BW: 48 bytes/cycle, 1.4 GHz
DRAM Schedule Queue	Size = 16 and Out of order (FR-RCFS)
Warp Scheduling Policy	Greedy then oldest (GTO)

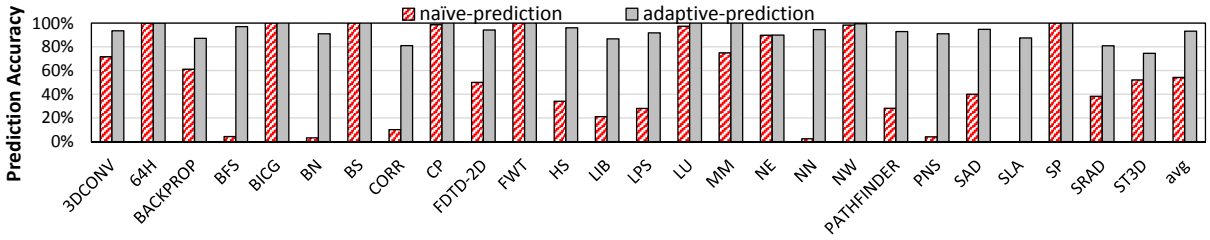

Figure 7.12: Prediction accuracy comparison between naive- and adaptive-prediction.

Benchmarks: In order to faithfully evaluate our proposed technique, we collect a large set of 26 representative applications from the NVIDIA CUDA SDK [31], PolyBench [98], Rodinia benchmark suite [32] and Parboil benchmark suite [33], with default inputs. We use a modified GPUWattch [17] to evaluate the dynamic and leakage energy consumption of LoSCache. We also include the energy consumption of our hardware extensions in our evaluation. In our experiments, all workloads run to completion on the simulator.

7.4 Results And Analysis

In this section, we first evaluate the prediction accuracy of LoSCache. We then analyze its impact on the interconnection network traffic and the overall performance. After that we explore the energy savings achieved by LoSCache. Finally, we conduct several sensitivity studies, including different L1 cache designs and warp scheduling policies, to evaluate the effectiveness of LoSCache under different design choices.

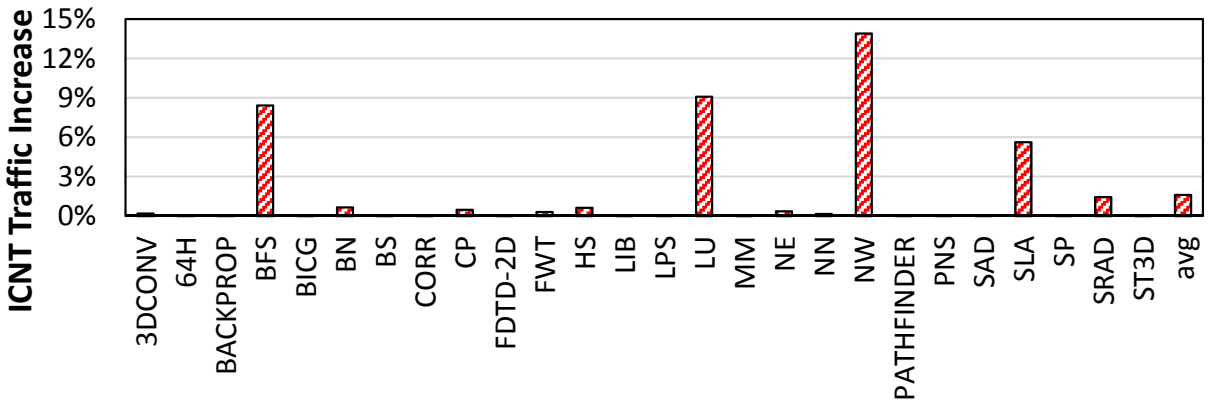

Figure 7.13: Normalized ICNT traffic increase over the baseline architecture.

7.4.1 Prediction Accuracy

We first evaluate the prediction accuracy of LoSCache. Figure 7.12 shows the prediction accuracy for naive- and adaptive-prediction (introduced in Section 7.2.3 to increase prediction accuracy), respectively. The prediction accuracy indicates the percentage of correct predictions (i.e., predicted access count is the same as the actual access count) among all the predictions. The figure shows that the prediction accuracy for the naive-prediction is low, which is 54% on average. In contrast, the adaptive prediction greatly increases the prediction accuracy to 93% on average. Based on our accuracy evaluation standard, we are checking if the prediction is absolutely accurate for each data. For some cases, the naive-prediction can only predict a similar locality but not the exact. However, by only adaptively increasing a small threshold, the prediction accuracy is significantly improved. For example, adding the threshold increases the prediction accuracy of `BFS` from 4% to 97%. On the other hand, the adaptive prediction is unable to further increase the prediction accuracy for some benchmarks, such as `64H`, `BICG`, `BS`, `CP`, `FWT`, `NE`, `SP`, since their prediction accuracy is already very high with the naive-prediction. Beyond this point, all the data collected from LoSCache design is using adaptive-prediction.

7.4.2 ICNT Traffic Increase

As discussion in Section 7.2.2, LoSCache uses the access counts of the L2 cache lines to set the access counts of the prediction table at the end of the prediction phase, which adds more interconnection network (ICNT) traffic. We further evaluate the ICNT traffic increase, shown in Figure 7.13. The average ICNT traffic increase is only 2% across all the benchmarks. The

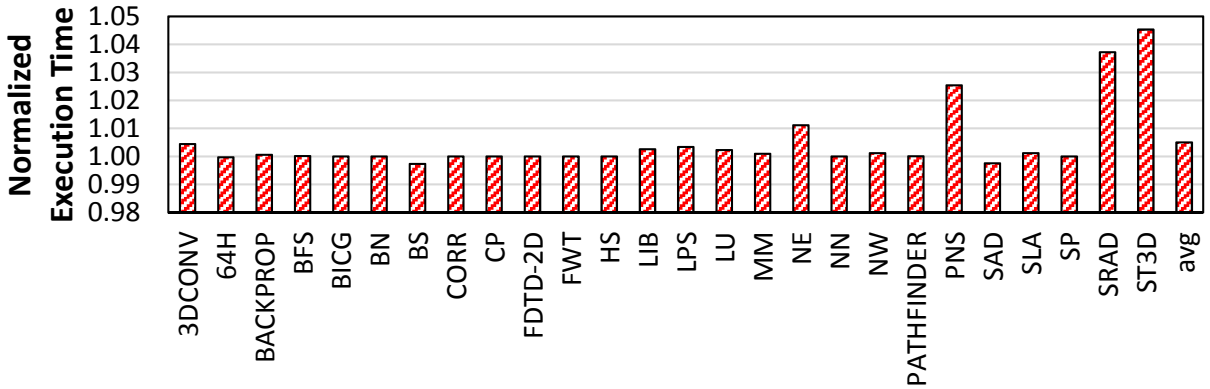

Figure 7.14: Normalized execution time of LoSCache against the baseline architecture.

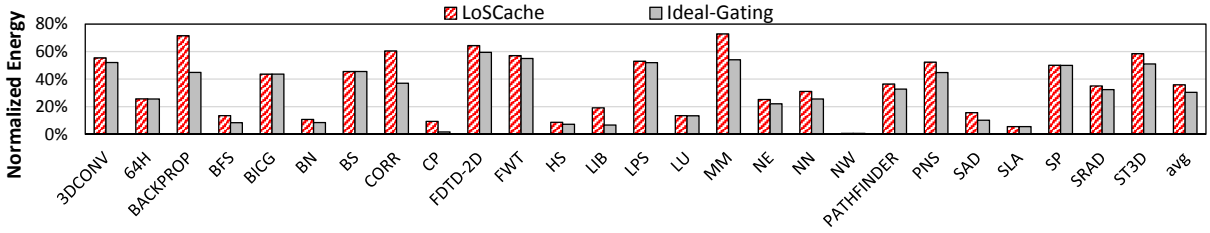

Figure 7.15: Normalized energy consumption for LoSCache and Ideal-Gating against the baseline architecture.

reason for this negligible ICNT traffic increase is due to the small size of the prediction tables. The average number of valid entries (i.e., entries that contain active prediction information) in a prediction table is only 7 across all the benchmarks (i.e., average number of the unique PCs in a table is 7). Also, LoSCache only updates the prediction tables at the end of prediction phase. Therefore, the ICNT traffic only increases by a small amount. Some Benchmarks, such as `BFS`, `LU`, `NW`, and `SLA`, have relatively large ICNT traffic increases, which are 8%, 9%, 14%, and 6%, respectively. It is due to their relatively large number of valid entries in the prediction tables (i.e., higher number of unique PCs) and small baseline ICNT traffic. For example, the number of valid entries in `NW` is 21 in each prediction table, while it only generates several hundred of interconnect network packages on the baseline architecture.

7.4.3 Performance Overhead

There are two main reasons for LoSCache to incur performance overhead: (1) the slight ICNT traffic increase at the end of the prediction phase (Figure 7.13), which may degrade the

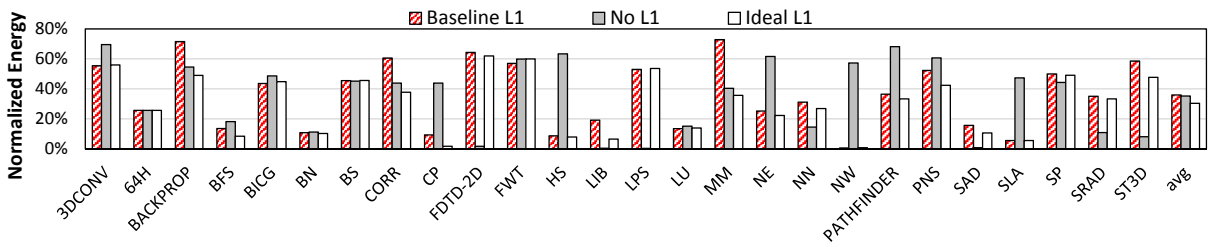

Figure 7.16: Normalized energy of LoSCache under no L1 data cache and ideal L1 data cache, respectively.

performance; and (2) cache lines suffer from the early power-gating due to inaccurate prediction, which increases L2 miss rate. Therefore, we further evaluate the execution time increase under LoSCache over the baseline architecture, as shown in Figure 7.14. The figure clearly demonstrates that LoSCache design does not noticeably affect the overall performance. On average, the execution time only increases by 0.5%. This result is consistent with the high prediction accuracy from our LoSCache design, shown in Figure 7.12. Also, even though the prediction accuracy for some benchmarks is lower, such as `ST3D` and `SRAD`, it only slightly hurts the performance. For example, the execution time of `ST3D` is only increased by 4%. This is because the small amount of early gating due to the incorrect prediction only slightly causes the performance degradation. Moreover, even though `NW` has a 14% ICNT traffic increase, its performance degradation is less than 1%. This is because its baseline ICNT traffic is very small, thus the traffic increase does not noticeably hurt the performance.

7.4.4 Energy Consumption

We also evaluate the energy consumption under LoSCache. Figure 7.15 shows the normalized energy results against the baseline architecture. As it shows, LoSCache greatly reduces the energy consumption by power gating the L2 cache lines during the dead time using a simple prediction mechanism. On average, the energy consumption is only 36% of the baseline case. LoSCache is effective for most benchmarks due to the low data re-reference counts in the L2 cache and its accurate prediction. For example, the energy consumption of `NW` is less than 1% of the baseline L2 cache due to its extremely low L2 cache utilization. On the other hand, LoSCache can only reduce the energy consumption of `MM` by 27% because of its high data re-reference counts. We also compare LoSCache with the Ideal-Gating technique, which is an

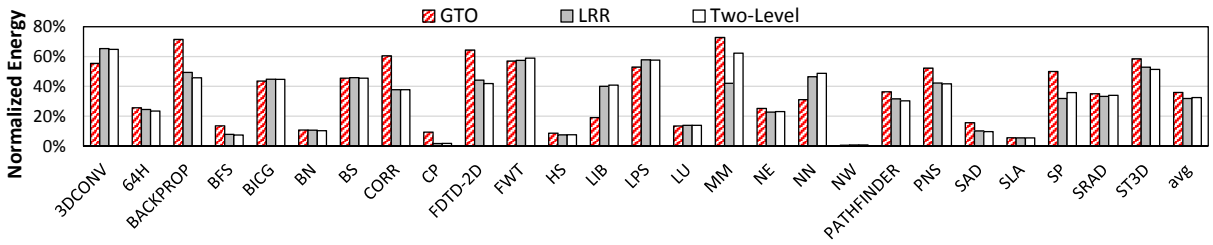

Figure 7.17: Normalized energy of LoSCache under different scheduling policies.

oracle case that all the L2 cache lines are powered off immediately once they expire. On average, Ideal-Gating further reduces the L2 energy consumption by 6% compared to LoSCache. The reasons are twofold. Although the prediction accuracy of LoSCache is high, it is not 100%. Thus it wastes a small portion of energy on storing the expired cache lines due to the inaccurate prediction. The other reason is that LoSCache cannot power gate any expired cache lines during the prediction phase. We further evaluate the energy saving of the entire GPU chip. Based on our evaluation, the energy consumption of GPU L2 cache is around 8% of the total GPU chip energy. As a consequence, LoSCache reduces 6.2% of the total GPU chip energy on average.

7.4.5 Sensitivity Analysis

The Impact of different L1 Cache Designs. We analyze the impact of different L1 cache designs on LoSCache, via evaluating two extreme L1 cache designs: no L1 cache and ideal L1 cache (i.e., very large to capture all the intra-CTA locality). Figure 7.16 shows the normalized energy consumption of L2 cache under these two schemes. To make the comparison more clear, we also include the results of LoSCache under the baseline L1 cache in the figure. Note that results in this figure are normalized to the baseline of each scheme (e.g., the result of LoSCache under no L1 case is normalized to the baseline L2 cache under no L1 cache). As it shows, LoSCache can greatly reduce the energy consumption under all three L1 cache designs. On average across all the benchmarks, the L2 cache energy under LoSCache is reduced to 36%, 35%, and 30% for baseline L1, no L1, and ideal L1, respectively. For many applications, including `BACKPROP`, `BFS`, `MM`, `PATHFINDER`, and `PNS`, LoSCache saves more energy under the ideal L1 cache design. This is because the ideal L1 cache design reduces the L2 cache utilization, thus the energy saving potential of LoSCache increases. On contrary, the No L1 design dramatically increases the L2 cache utilization, thus the energy saving potential of LoSCache decreases.

However, we also observe some unconventional results, where the energy savings are larger for some benchmarks under No L1 design, including `FDTD-2D`, `LIB`, `LPS`, `NN`, `SAD`, `SRAD` and `ST3D`. This is because LoSCache mainly reduces the leakage energy of the L2 cache. Under No L1 design, the total program execution time increases for applications with good L1 cache locality, thus the leakage portion of the L2 cache energy increases. As a consequence, LoSCache under No L1 design saves more energy. The above also confirms that our design is complement to the state-of-the-art L1 cache designs to conserve additional energy.

The Impact of Scheduling Policies. We further analyze the impact of different warp scheduling policies on LoSCache. Figure 7.17 shows the results of Greedy-Then-Oldest (GTO) [12], Loose-Round-Robin (LRR), and Two-Level [99] scheduling. As it shows, on average across all the benchmarks, the energy consumption of LoSCache, is reduced to 36%, 32%, and 32%, respectively. It indicates that the energy saving ability of LoSCache is very effective under different scheduling policies. We also observe that LoSCache achieves lower energy savings under GTO scheduling. This is because the the execution progress of different warps has the largest variations under GTO, which hurts the prediction accuracy of LoSCache.

Chapter 8

Related Work

8.1 Soft-Error Protection Mechanisms in GPUs

There have been works on investigating, detecting, and recovering the soft errors on GPUs [100, ?, 101, 21, 102, 103, 104, 105, 6, 106]. Sheaffer et al. [103] explore the concept of the sphere of replication on GPU processors, and present a hardware redundancy-based approach to create a reliable GPU with no performance loss. Dimitrov [104] investigate three software approaches to perform redundant execution for GPU reliability. Maruyama et al. [5] propose a high-performance software framework to enhance GPU with DRAM fault tolerance. It leverages light-weight data coding for error detection and checkpointing for recovery. Solano-Quinde et al. [105] propose an application-level checkpoint scheme for GPU systems, and explore the computation-communication overlapping to reduce the checkpoint overhead. In our study, we exploit the SPs idle time and recycle it to perform the redundancy for error detection to maximize the SPs reliability with little performance loss. Recently, Nathan et al. [107] develop Argus-G, it implements control flow, dataflow and computation checkers in the SPs for low cost error detection. Yim et al. [6] strategically places customized error detectors in the source code of GPU applications to tolerate errors. Both the two schemes are orthogonal to our proposed RISE and LESS techniques.

8.2 STT-RAM based Architecture Design

STT-RAM has been widely investigated as a replacement for SRAM based structures [40, 41, 42]. For instance, Smullen et al. [41] redesign the STT-RAM cell to improve its write time and energy by trading-off the retention time. Besides using STT-RAM to build stand-alone memory, there have been many studies integrating STT-RAM into both CPU and GPUs microarchitecture design [42, 35]. For example, Guo et al. [42] explore storage and combinational

logic with STT-RAM to resist the power wall of CMPs. All these studies focus on the high density, low leakage features of STT-RAM to deliver energy-efficient CPUs/GPUs. And the studies of using STT-RAM soft-error resilience for reliable computing limit in CPUs [37, 38]. To our knowledge, LESS the first work to use STT-RAM combined with the GPGPU unique characteristics to build soft-error robust and low-power GPUs RF.

8.3 Mitigating the Impact of Process Variations in GPUs

There have been several studies on analyzing and mitigating the PV impact on GPUs. For instance, Jing et al. [16] modeled the effect of PV on register file that is built based on SRAM and eDRAM, respectively. However, this work mainly focuses on the variation of RF retention time and energy under PV, and does not perform a detailed analysis of the frequency degradation. Lee et al. [62] observed the large frequency variations across SMs, and proposed the PV mitigation techniques at the SM level that running each SM at its maximum frequency independently and disabling the slowest SM. Aguilera et al. [108] proposed PV-aware SM partitioning for applications in GPUs that supports spatial-multitasking. Krimer et al. [79, 109] investigated the timing errors under the impact of process variations. Since an error in any SPMD lane stalls the entire pipeline, they further proposed lane decoupling that enables each SPMD lane to tolerate timing errors independent of other adjacent lanes, therefore, boosting the GPUs throughput. Rahimi et al. [110] further proposed to alleviate the cost of timing error recovery by exploiting the value locality inside parallel programs, saving the result of an error-free lane, and further using it to spatially correct faulty instruction in error-prone lanes. As a conservative method to mitigate the PV impact, designers can set a frequency guardband to ensure that processors run correctly during its expected lifetime. As its disadvantage, guardbanding causes high frequency loss. Rahimi et al. [111] proposed the hierarchically focused guardbanding technique on GPUs that monitors PV at fine-grain instruction-level and coarse-grain kernel-level and dynamically adapts guardbanding to gain performance optimization. Our PV mitigation techniques are orthogonal to those previously proposed mechanisms.

8.4 Enhancing the Reliability of Storage Cells

Several architecture and circuit level solutions have been proposed to improve the reliability of storage cells in cache and memory. For instance, the faulty entries in cache and memory can be handled via disabling their faulty cells at different granularities [76, 81, 82, 83, 84, 85]. As a side-effect, these methods result in nontrivial cache/memory capacity reduction. Since the GPU register file is a statically allocated resource and is critical to the computational throughput (or TLP), reducing its capacity could result in concurrency reduction, which directly affects performance, and may even fail to execute applications that require high register usage. One recently proposed technique [85] leverages the redundant data copies in other hardware structures (e.g., MSHRs, micro-op cache, and store queue) to patch the faulty subblocks in the cache. When the required redundant data copies are not found, the cache accesses are treated as misses to the lower memory hierarchy. However, this opportunistic fault-patching technique is not applicable to the GPU register file. If the correct data copies from the other hardware components do not exist upon request, treating accesses to the faulty register entries as misses could result in program crash. In contrast, our GR-Guard technique leverages the unique register dead opportunity in GPGPUs applications to enhance the register file reliability under process variations and low supply voltages.

8.5 Improving the Efficiency of Caches

Due to the massive thread-level parallelism and the limited capacity, the GPU caches are not efficiently used. Several recent work has addressed this problem to mitigate the performance loss due to cache conflicts [12, 13, 91, 92, 93, 94, 95, 96]. For example, Rogers et al. [12] proposed a cache-conscious wavefront scheduling to reduce the number of warp that can access the L1 cache, thus the intra-CTA cache locality is better maintained and performance is improved. Xie et al. [91] proposed to improve the cache performance via a coordinated static code analysis and dynamic cache bypassing. These techniques all effectively improve the L1 cache performance by exploiting the data locality and avoiding the cache thrashing. In this way, the data locality is well maintained in L1 cache. However, the utilization of L2 cache in these designs remains low. In contrast, our proposed LoSCache technique addresses the inefficiency of the L2 cache

in GPU by leveraging the instruction-level data locality similarity to reduce the unnecessary energy waste. LoSCache is compatible with all these L1 cache management techniques and can achieve further energy savings. Recently, Ausavarungnirun et al. [112] proposed a memory divergence correction technique to deal with the memory divergence and cache queuing problem in L2 cache. Their work improves the energy-efficiency via cache bypassing and insertion. In contrast, our LoSCache improves the energy-efficiency through re-reference prediction. Their technique is orthogonal to ours.

Several previous work was also proposed to reduce the energy consumption of CPU caches [113, 114, 115, 116, 117]. For example, Kaxiras et al. proposed to power off a cache line if it is not accessed after a long period of time [113]. For this technique, the prediction for cache line dead-time is coarse-grained and only based on the cache line itself. In contrast, our technique leverages the unique locality similarity of GPU programs to accurately predict the dead-time in advance and powers off a cache line immediately after the last request of a data access. To reduce its prediction overhead, Khan et al. [117] observed the consistent memory access patterns across cache sets and used a small number of cache lines for prediction. However based on our observation, GPU L2 cache accesses are irregular and do not have this consistent pattern, thus this technique is not applicable to the GPU architecture. Additionally, several other work has been proposed to reduce the energy consumption of GPU caches. For example, Wang et al. propose to put the L1 and L2 caches to sleep mode when they are not active to save energy [118]. This work reduces the energy when the entire cache is not active. In contrast, our LoSCache is able to reduce the L2 energy when it is active, by predicting the cache dead-time. Furthermore, some work also proposed to leverage emerging hardware technologies such as STT-RAM [37] and domain wall memory [119] to build power-efficient GPU cache. Our LoSCache is orthogonal to these work as well.

Chapter 9

Conclusion

With their strong computing power and improved programmability, GPUs emerge as highly-efficient devices for a wide range of parallel applications. As the process technology scales down in recent years, reliability and energy-efficiency become severer and are growing threats to modern GPUs.

In this dissertation, we first propose RISE to effectively recycle the SPs idle time for soft-error detection, which is the first essential step to optimize the soft-error robustness for the GPUs. RISE is composed of full-RISE and partial-RISE. Full-RISE exploits the fully idled SPs caused by the long-latency memory transactions and imbalanced load assignment among GPU cores, and uses them to perform the redundant execution and enhance the SPs reliability. Partial-RISE combines the redundant execution of a number of threads from a warp with a diverged warp to recycle the SPs idle time during the branch divergence for their reliability optimization. Experiment results show that RISE reduces the SPs AVF by 43% with only 4% performance loss. Sensitivity analysis also shows that RISE is applicable to GPUs with various performance optimization mechanisms.

We then propose an energy-efficient protection mechanism for the vulnerable and power-hungry register file. STT-RAM exhibits the benefit of low leakage and high density, and more interestingly, it is immune to the radiation induced soft errors. In this work, we propose LESS, which leverages STT-RAM and the unique characteristics of GPGPU applications to explore the soft-error robust and low-power RF in GPUs without hurting the performance. LESS is composed of two mechanisms: LA-LESS that designs a SRAM and STT-RAM hybrid RF and protects the long-lived register values via STT-RAM RF; and NWA-LESS that combines two narrow-width STT-RAM writes. LESS achieves 86% RF soft-error vulnerability improvement, 60% RF energy savings with only 4% performance loss.

We also propose to mitigate the susceptibility of GPUs register file to PV. We first propose to vertically divide RF banks into sub-banks, and explore the variable-latency technique at sub-bank level to perform the coarse-grain register classification which maximizes the frequency optimization under PV. We then propose to re-organize RF banks by virtually combining the same-type sub-banks, leading to the fast and slow RF bank categories. The IPC degrades when using the slow RF banks during the kernel execution. We further explore two techniques that leverage the unique features in GPGPUs application to mitigate the IPC loss: the FWAS (fast-warp aware warp scheduling) technique helps to minimize the use of slow RF banks; and the hybrid bank technique allocates the partially used register vectors into the fast sub-bank portions of the virtually built hybrid RF banks and eliminate the negative impact of their slow sub-bank portions on performance. These PV mitigation techniques obtains 15% overall performance improvement compared to the baseline case without any optimization.

We further address the reliability challenge of GPU register file under aggressive voltage reduction and process variations, which can enable further energy savings beyond the existing techniques such as voltage noise smoothing and power gating. As the largest on-chip storage structure, register file becomes the reliability hotspot under voltage reduction. We propose an effective architectural solution named GR-Guard, that leverages the inherent long register dead time on GPU to enable reliable operations at low voltages with negligible performance overhead. This work serves as an essential step for enabling chip-wide supply voltage reduction for future GPU design. Experimental results show GR-Guard can enable reliable GPU register file with less than 2% performance degradation under aggressive voltage reduction, while achieving an average of 31% energy reduction.

Finally, we address the energy-inefficiency of GPU L2 cache. L2 cache is underutilized in the modern GPUs. It stores useless data for the majority of time since GPU applications lack of inter-CTA locality. As a consequence, L2 data has no or few future re-references. To address this inefficient utilization of L2 cache, we propose LoSCache, a simple and effective energy-efficient L2 design. LoSCache leverages the unique instruction-level data locality similarity caused by the SIMT programming model in GPUs to predict the data re-reference counts in L2. It further powers off the L2 cache lines if they are predicted to be dead after certain accesses. Experimental

results show that LoSCache dramatically reduces the energy consumption of the L2 cache by 64% with only 0.5% performance overhead. Sensitivity studies also demonstrate that LoSCache is compatible with the state-of-the-art L1 cache designs and can achieve further energy savings.

Bibliography

[1] NVIDIA, "Cuda C Programming Guide." http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2015.

[2] AMD, "Amd Accelerated Parallel Processing: OpenCL Programming Guide." http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf, 2015.

[3] "Restore: Symptom Based Soft Error Detection in Microprocessors," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, (Washington, DC, USA), pp. 30–39, IEEE Computer Society, 2005.

[4] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, "Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, (Washington, DC, USA), pp. 264–, IEEE Computer Society, 2004.

[5] N. Maruyama, A. Nukada, and S. Matsuoka, "A High-Performance Fault-Tolerant Software Framework for Memory on Commodity GPUs," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, April 2010.

[6] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight Silent Data Corruption Error Detector for GPGPU," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pp. 287–300, May 2011.

[7] R. B. P. B. V. J. R. Jingwen Leng, Alper Buyuktosunoglu, "Safe Limits on Voltage Reduction Efficiency in Gpus: a Direct Measurement Approach," in *in Proceedings of the IEEE International Symposium On Microarchitecture (MICRO)*, Dec 2015.

[8] S. Borkar, "Design Challenges of Technology Scaling," *Micro, IEEE*, vol. 19, pp. 23–29, Jul 1999.

[9] J. Tan and X. Fu, "Rise: Improving the Streaming Processors Reliability against Soft Errors in GPGPUs," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 191–200, ACM, 2012.

[10] J. Tan, Z. Li, and X. Fu, "Soft-Error Reliability and Power Co-Optimization for GPGPUs Register File Using Resistive Memory," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, (San Jose, CA, USA), pp. 369–374, EDA Consortium, 2015.

[11] J. Tan and X. Fu, "Mitigating the Susceptibility of GPGPUs Register File to Process Variations," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pp. 969–978, May 2015.

[12] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 72–83, IEEE Computer Society, 2012.

[13] X. Chen, L. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W. Hwu, "Adaptive Cache Management for Energy-Efficient GPU Computing," in *Proceedings of the 47th Annual*

IEEE/ACM International Symposium on Microarchitecture, MICRO-47, (Washington, DC, USA), pp. 343–355, IEEE Computer Society, 2014.

[14] I. Singh, A. Shriraman, W. Fung, M. O'Connor, and T. Aamodt, "Cache Coherence for GPU Architectures," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 578–590, Feb 2013.

[15] W. Jia, K. Shaw, and M. Martonosi, "MRPB: Memory Request Prioritization for Massively Parallel Processors," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 272–283, Feb 2014.

[16] N. Jing, Y. Shen, Y. Lu, S. Ganapathy, Z. Mao, M. Guo, R. Canal, and X. Liang, "An Energy-Efficient and Scalable Edram-Based Register File Architecture for GPGPU," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 344–355, ACM, 2013.

[17] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling Energy Optimizations in GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 487–498, ACM, 2013.

[18] M. Abdel-Majeed and M. Annavaram, "Warped Register File: a Power Efficient Register File for GPGPUs," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, (Washington, DC, USA), pp. 412–423, IEEE Computer Society, 2013.

[19] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 163–174, April 2009.

[20] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-Efficient Mechanisms for Managing Thread Context in Throughput Processors," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 235–246, ACM, 2011.

[21] S. K. Reinhardt and S. S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, (New York, NY, USA), pp. 25–36, ACM, 2000.

[22] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 308–317, ACM, 2011.

[23] X. Yang, X. Liao, W. Xu, J. Song, Q. Hu, J. Su, L. Xiao, K. Lu, Q. Dou, J. Jiang, and C. Yang, "Th-1: China'S First Petaflop Supercomputer," *Front. Comput. Sci China*, vol. 4, pp. 445–455, Dec. 2010.

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

[25] M. A. Gomaa and T. N. Vijaykumar, "Opportunistic Transient-Fault Detection," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, (Washington, DC, USA), pp. 172–183, IEEE Computer Society, 2005.

[26] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2010.

[27] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pp. 407–420, Dec 2007.

[28] W. W. L. Fung and T. M. Aamodt, "Thread Block Compaction for Efficient Simt Control Flow," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, (Washington, DC, USA), pp. 25–36, IEEE Computer Society, 2011.

[29] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, (Washington, DC, USA), pp. 29–, IEEE Computer Society, 2003.

[30] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[31] NVIDIA, "Nvidia CUDA Sdk." https://developer.nvidia.com/cuda-downloads, 2015.

[32] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: a Benchmark Suite for Heterogeneous Computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, Oct 2009.

[33] I. J. S. N. O. L. C. N. A. G. D. L. W. W. H. John A. Stratton, Christopher Rodrigues, "Parboil: a Revised Benchmark Suite for Scientific and Commercial Throughput Computing," tech. rep., University of Illinois at Urbana-Champaign, 03 2012.

[34] N. B. Lakshminarayana and H. Kim, "Effect of Instruction Fetch and Memory Scheduling on GPU Performance," in *Workshop on Language, Compiler, and Architecture Support for GPGPU, in conjunction with HPCA/PPoPP*, 2010.

[35] N. Goswami, B. Cao, and T. Li, "Power-Performance Co-Optimization of Throughput Core Architecture Using Resistive Memory," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 342–353, Feb 2013.

[36] P. Satyamoorthy, *STT-RAM for Shared Memory in GPUs.* PhD thesis, University of Virginia, 2011.

[37] H. Sun, C. Liu, W. Xu, J. Zhao, N. Zheng, and T. Zhang, "Using Magnetic RAM to Build Low-Power and Soft Error-Resilient L1 Cache," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 20, pp. 19–28, Jan. 2012.

[38] G. Sun, E. Kursun, J. Rivers, and Y. Xie, "Exploring the Vulnerability of CMPS to Soft Errors with 3D Stacked Non-Volatile Memory," in *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pp. 366–372, Oct 2011.

[39] Freescale, "Document Number Brmramslscltrl," *Freescale MRAM technology*, 2007.

[40] Z. Sun, X. Bi, H. H. Li, W. Wong, Z. Ong, X. Zhu, and W. Wu, "Multi Retention Level STT-RAM Cache Designs with a Dynamic Refresh Scheme," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 329–338, ACM, 2011.

[41] C. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. Stan, "Relaxing Non-Volatility for Fast and Energy-Efficient STT-RAM Caches," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 50–61, Feb 2011.

[42] X. Guo, E. Ipek, and T. Soyata, "Resistive Computation: Avoiding the Power Wall with Low-Leakage, STT-MRAM Based Computing," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 371–382, ACM, 2010.

[43] C. W. Smullen, IV, *Designing Giga-scale Memory Systems with STT-RAM*. PhD thesis, Charlottesville, VA, USA, 2011.

[44] M. Gebhart, S. W. Keckler, and W. J. Dally, "A Compile-Time Managed Multi-Level Register File Hierarchy," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 465–476, ACM, 2011.

[45] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, "Using Datalog with Binary Decision Diagrams for Program Analysis," in *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, (Berlin, Heidelberg), pp. 97–118, Springer-Verlag, 2005.

[46] D. Brooks and M. Martonosi, "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance," in *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pp. 13–22, Jan 1999.

[47] S. Gilani, N. S. Kim, and M. Schulte, "Power-Efficient Computing for Compute-Intensive GPGPU Applications," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 330–341, Feb 2013.

[48] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A Novel Architecture of the 3D Stacked Mram L2 Cache for CMPS," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pp. 239–249, Feb 2009.

[49] S. Wilton and N. Jouppi, "Cacti: an Enhanced Cache Access and Cycle Time Model," *Solid-State Circuits, IEEE Journal of*, vol. 31, pp. 677–688, May 1996.

[50] X. Dong, C. Xu, Y. Xie, and N. Jouppi, "Nvsim: a Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 31, pp. 994–1007, July 2012.

[51] NVIDIA, "Nvidia'S Next Generation CUDA Computer Architecture: Fermi." http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf, 2015.

[52] P. Montesinos, W. Liu, and J. Torrellas, "Using Register Lifetime Predictions to Protect Register Files against Soft Errors," in *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pp. 286–296, June 2007.

[53] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "Mcpat: an Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 469–480, Dec 2009.

[54] M. T. Chang, P. Rosenfeld, S. L. Lu, and B. Jacob, "Technology Comparison for Large Last-Level Caches (L3Cs): Low-Leakage SRAM, Low Write-Energy STT-RAM, and Refresh-Optimized Edram," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 143–154, Feb 2013.

[55] D. Kanter, "Amd'S Cayman GPU Architecture." http://www.realworldtech.com/cayman/, 2015.

[56] NVIDIA, "Nvidia'S Next Generation CUDA Computer Architecture: Kepler." http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2015.

[57] J. Tschanz, J. Kao, S. Narendra, R. Nair, D. Antoniadis, A. Chandrakasan, and V. De, "Adaptive Body Bias for Reducing Impacts of Die-To-Die and Within-Die Parameter Variations on Microprocessor Frequency and Leakage," *Solid-State Circuits, IEEE Journal of*, vol. 37, pp. 1396–1402, Nov 2002.

[58] A. Agarwal, B. Paul, H. Mahmoodi, A. Datta, and K. Roy, "A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 13, pp. 27–38, Jan 2005.

[59] S. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, "Varius: a Model of Process Variation and Resulting Timing Errors for Microarchitects," *Semiconductor Manufacturing, IEEE Transactions on*, vol. 21, pp. 3–13, Feb 2008.

[60] U. R. Karpuzcu, K. B. Kolluru, N. S. Kim, and J. Torrellas, "Varius-Ntv: a Microarchitectural Model to Capture the Increased Sensitivity of Manycores to Process Variations at Near-Threshold Voltages," in *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2012.

[61] A. Agrawal, A. Ansari, and J. Torrellas, "Mosaic: Exploiting the Spatial Locality of Process Variation to Reduce Refresh Energy in On-Chip Edram Modules," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 84–95, Feb 2014.

[62] J. Lee, P. Ajgaonkar, and N. S. Kim, "Analyzing Throughput of GPGPUs Exploiting Within-Die Core-To-Core Frequency Variation," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pp. 237–246, April 2011.

[63] X. Liang and D. Brooks, "Mitigating the Impact of Process Variations on Processor Register Files and Execution Units," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pp. 504–514, Dec 2006.

[64] R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, "Mitigating Parameter Variation with Dynamic Fine-Grain Body Biasing," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pp. 27–42, Dec 2007.

[65] S. Seo, R. Dreslinski, M. Woh, Y. Park, C. Charkrabari, S. Mahlke, D. Blaauw, and T. Mudge, "Process Variation in Near-Threshold Wide SIMD Architectures," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pp. 980–987, June 2012.

[66] Y. Pan, J. Kong, S. Ozdemir, G. Memik, and S. W. Chung, "Selective Wordline Voltage Boosting for Caches to Manage Yield under Process Variations," in *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, (New York, NY, USA), pp. 57–62, ACM, 2009.

[67] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: a Tool to Model Large Caches," *HP Laboratories*, pp. 22–31, 2009.

[68] M. Yoshimoto, K. Anami, H. Shinohara, T. Yoshihara, H. Takagi, S. Nagao, S. Kayano, and T. Nakano, "A Divided Word-Line Structure in the Static RAM and Its Application to a 64K Full CMOS RAM," *Solid-State Circuits, IEEE Journal of*, vol. 18, pp. 479–485, Oct 1983.

[69] M. Tehranipour, Z. Navabi, and S. Fakhraie, "An Efficient BIST Method for Testing of Embedded Srams," in *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, vol. 5, pp. 73–76 vol. 5, 2001.

[70] M. Rhu and M. Erez, "Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 356–367, ACM, 2013.

[71] J. Leng, Y. Zu, and V. Reddi, "GPU Voltage Noise: Characterization and Hierarchical Smoothing of Spatial and Temporal Voltage Noise Interference in GPU Architectures," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 161–173, Feb 2015.

[72] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-Compression: Enabling Power Efficient GPUs through Register Compression," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 502–514, ACM, 2015.

[73] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 86–98, ACM, 2013.

[74] W.-k. S. Yu, R. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Suh, "SRAM-DRAM Hybrid Memory with Applications to Efficient Register Files in Fine-Grained Multi-Threading," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 247–258, ACM, 2011.

[75] M. Abdel-Majeed, D. Wong, and M. Annavaram, "Warped Gates: Gating Aware Scheduling and Power Gating for GPGPUs," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 111–122, ACM, 2013.

[76] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, "Trading off Cache Capacity for Reliability to Enable Low Voltage Operation," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, (Washington, DC, USA), pp. 203–214, IEEE Computer Society, 2008.

[77] A. Bacha and R. Teodorescu, "Dynamic Reduction of Voltage Margins by Leveraging On-Chip ECC in Itanium III Processors," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 297–307, ACM, 2013.

[78] A. Bacha and R. Teodorescu, "Using ECC Feedback to Guide Voltage Speculation in Low-Voltage Processors," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 306–318, Dec 2014.

[79] E. Krimer, P. Chiang, and M. Erez, "Lane Decoupling for Improving the Timing-Error Resiliency of Wide-Simd Architectures," in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pp. 237–248, June 2012.

[80] NVIDIA, "Nvidia Geforce Gtx 980: Featuring Maxwell, the Most Advanced GPU Ever Made.." http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF, 2015.

[81] P. J. Nair, D. H. Kim, and M. K. Qureshi, "Archshield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates," in *Proceedings of the 40th*

Annual International Symposium on Computer Architecture, ISCA '13, (New York, NY, USA), pp. 72–83, ACM, 2013.

[82] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. Jouppi, and M. Erez, "Free-P: Protecting Non-Volatile Memory against Both Hard and Soft Errors," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 466–477, Feb 2011.

[83] A. Ansari, S. Feng, S. Gupta, and S. Mahlke, "Archipelago: a Polymorphic Cache Design for Enabling Robust Near-Threshold Operation," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 539–550, Feb 2011.

[84] A. Ansari, S. Gupta, S. Feng, and S. Mahlke, "Zerehcache: Armoring Cache Architectures in High Defect Density Technologies," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 100–110, Dec 2009.

[85] D. Palframan, N. S. Kim, and M. Lipasti, "Ipatch: Intelligent Fault Patching to Improve Energy Efficiency," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 428–438, Feb 2015.

[86] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. Hoe, "Multi-Bit Error Tolerant Caches Using Two-Dimensional Error Coding," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pp. 197–209, Dec 2007.

[87] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 72–83, IEEE Computer Society, 2012.

[88] A. W. Appel, *Modern Compiler Implementation in C: Basic Techniques*. New York, NY, USA: Cambridge University Press, 1997.

[89] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler, "A Variable Warp Size Architecture," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 489–501, ACM, 2015.

[90] R. Foundation, "R: the R Project for Statistical Computing." https://www.r-project.org/, 2015.

[91] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated Static and Dynamic Cache Bypassing for GPUs," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 76–88, Feb 2015.

[92] D. Li, M. Rhu, D. Johnson, M. O'Connor, M. Erez, D. Burger, D. Fussell, and S. Redder, "Priority-Based Cache Allocation in Throughput Processors," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 89–100, Feb 2015.

[93] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-Driven Dynamic GPU Cache Bypassing," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, (New York, NY, USA), pp. 67–77, ACM, 2015.

[94] A. Li, G. J. van den Braak, A. Kumar, and H. Corporaal, "Adaptive and Transparent Cache Bypassing for GPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, (New York, NY, USA), pp. 17:1–17:12, ACM, 2015.

[95] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Owl: Cooperative Thread Array Aware Scheduling

Techniques for Improving GPGPU Performance," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, (New York, NY, USA), pp. 395–406, ACM, 2013.

[96] Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jiménez, "Adaptive GPU Cache Bypassing," in *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, (New York, NY, USA), pp. 25–35, ACM, 2015.

[97] S. Y. Lee and C. J. Wu, "Caws: Criticality-Aware Warp Scheduling for GPGPU Workloads," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), pp. 175–186, ACM, 2014.

[98] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-Tuning a High-Level Language Targeted to GPU Codes," in *Innovative Parallel Computing (InPar), 2012*, pp. 1–10, May 2012.

[99] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-Efficient Mechanisms for Managing Thread Context in Throughput Processors," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 235–246, ACM, 2011.

[100] I. Haque and V. Pande, "Hard Data on Soft Errors: a Large-Scale Assessment of Real-World Error Rates in GPGPU," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pp. 691–696, May 2010.

[101] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing Soft-Error Vulnerability on GPGPU Microarchitecture," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pp. 226–235, Nov 2011.

[102] N. Maruyama, A. Nukada, and S. Matsuoka, "Software-Based ECC for GPUs," in *Symposium on Application Accelerators in High-Performance Computing*, SAAHPC '09, 2009.

[103] J. W. Sheaffer, D. P. Luebke, and K. Skadron, "A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors," in *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, (Aire-la-Ville, Switzerland, Switzerland), pp. 55–64, Eurographics Association, 2007.

[104] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding Software Approaches for GPGPU Reliability," in *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, (New York, NY, USA), pp. 94–104, ACM, 2009.

[105] L. Solano-Quinde, B. Bode, and A. Somani, "Coarse Grain Computation-Communication Overlap for Efficient Application-Level Checkpointing for GPUs," in *Electro/Information Technology (EIT), 2010 IEEE International Conference on*, pp. 1–5, May 2010.

[106] D. Palframan, N. S. Kim, and M. Lipasti, "Precision-Aware Soft Error Protection for GPUs," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 49–59, Feb 2014.

[107] R. Nathan and D. J. Sorin, "Argus-G: Comprehensive, Low-Cost Error Detection for GPGPU Cores," *IEEE Computer Architecture Letters*, vol. 14, pp. 13–16, Jan 2015.

[108] P. Aguilera, J. Lee, A. Farmahini-Farahani, K. Morrow, M. Schulte, and N. S. Kim, "Process Variation-Aware Workload Partitioning Algorithms for GPUs Supporting Spatial-Multitasking," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pp. 1–6, March 2014.

[109] E. Krimer, R. Pawlowski, M. Erez, and P. Chiang, "Synctium: a Near-Threshold Stream Processor for Energy-Constrained Parallel Applications," *Computer Architecture Letters*, vol. 9, pp. 21–24, Jan 2010.

[110] A. Rahimi, L. Benini, and R. Gupta, "Spatial Memoization: Concurrent Instruction Reuse to Correct Timing Errors in SIMD Architectures," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 60, pp. 847–851, Dec 2013.

[111] A. Rahimi, L. Benini, and R. K. Gupta, "Hierarchically Focused Guardbanding: an Adaptive Approach to Mitigate PVT Variations and Aging," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, (San Jose, CA, USA), pp. 1695–1700, EDA Consortium, 2013.

[112] R. Ausavarungnirun, S. Ghose, O. Kayıran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu, "Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance," in *Proceedings of the The 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT'15, 2015.

[113] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power," in *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pp. 240–251, 2001.

[114] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache Bursts: a New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, (Washington, DC, USA), pp. 222–233, IEEE Computer Society, 2008.

[115] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior," in *Computer Architecture, 2002. Proceedings. 29th*

Annual International Symposium on, pp. 209–220, 2002.

[116] A. C. Lai, C. Fide, and B. Falsafi, "Dead-Block Prediction; Dead-Block Correlating Prefetchers," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, (New York, NY, USA), pp. 144–154, ACM, 2001.

[117] S. Khan, Y. Tian, and D. Jimenez, "Sampling Dead Block Prediction for Last-Level Caches," in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pp. 175–186, Dec 2010.

[118] Y. Wang, S. Roy, and N. Ranganathan, "Run-Time Power-Gating in Caches of GPUs for Leakage Energy Savings," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, (San Jose, CA, USA), pp. 300–303, EDA Consortium, 2012.

[119] R. Venkatesan, S. Ramasubramanian, S. Venkataramani, K. Roy, and A. Raghunathan, "Stag: Spintronic-Tape Architecture for GPGPU Cache Hierarchies," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pp. 253–264, June 2014.