Copyright
by
Minsoo Rhu
2014

The Dissertation Committee for Minsoo Rhu
certifies that this is the approved version of the following dissertation:

Performance-Efficient Mechanisms for Managing Irregularity in Throughput Processors

Committee:

Mattan Erez, Supervisor

Donald S. Fussell

Andreas Gerstlauer

Vijay Janapa Reddi

Stephen W. Keckler

Performance-Efficient Mechanisms for Managing Irregularity in Throughput Processors

by

Minsoo Rhu, B.E.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2014

Dedicated to my wife, Hoili Lee

and my loving parents, Hoo-Kyu Rhu and Soon-Ja Kim.

Acknowledgments

First, I would like express my sincere gratitude to my advisor Mattan Erez. When I first came to Austin as a Ph.D. student, I knew very little about computer architecture, but Mattan has patiently teached me through all the aspects of what it takes to do high quality computer architecture research. He was always a caring, supportive, yet rigorous advisor whom I can always trust and ask for advice. Thanks to Mattan's guidance and his high standard, I had the privilege of presenting a number of my research ideas at top-notch computer architecture conferences, none of which would have been possible had I not have Mattan as my advisor. I truly thank him for his help and support in making this dissertation possible.

I would also like to thank the other members of my dissertation committee: Professor Donald Fussell, Professor Andreas Gerstlauer, Professor Vijay Reddi, and Dr. Stephen Keckler. Their keen insights and feedbacks have been invaluable in developing and elaborating the concepts in my dissertation. Special thanks to Professor Vijay Reddi whom I had the pleasure to collaborate with during the last two semesters of my time at Austin. Vijay gave me great insights on getting my dissertation squared away and also provided me with invaluable advice on my research as well as my job hunts.

I had an opportunity to collaborate with the researchers at NVIDIA Research during the summer of 2013 as a research intern. I would like to take this opportunity to thank my manager Stephen Keckler, who also served on my dissertation committee, as well as my mentors Daniel Johnson, Mike O'Connor and Jason Clemons. They helped me better understand various aspects of the state-of-the-art GPU architectures and I feel fortunate for such an opportunity of collaboration.

I should also thank my friends at the University of Texas at Austin: Jinsuk Chung, Seong-lyong Gong, Jingwen Leng, Yuhao Zhu, Yunjin Ko, Jeongmin Hyun, Hyokyung Kim, Jaeyoon Lee, Minhye Shin, Junyoung Park, Donghyuk Shin, and Joonhyoung Ro. Special thanks to Jinsuk, Seong-lyong, and Jingwen for their friendship throughout my doctoral study at Austin. Jinsuk and Seong-lyong have been great friends since the first day I arrived at Austin, making Austin feel at home and giving me great advice not only in terms of research but also in life. I first became friends with Jingwen during our class project on building cache simulators, which later led to our collaboration on our very first MICRO paper. The MICRO paper would have never made it without Jingwen's expertise on power modeling and I thank him for his help as well as his friendship. I also thank the members of our LPH (Locality, Parallelism, and Hierarchy) group: Ikhwan Lee, Min Kyu Jeong, Mike Sullivan, Song Zhang, Dong Wan Kim, Jungrae Kim, Tianhao Zheng, Haishan Zhu, Derong Liu, and Kyushik Lee. I will miss all the exciting discussions we had as well as all the pizzas we ate during our group meetings.

In addition to the folks I met through my years in Austin, I also want to thank the people back in Korea who always welcomed me whenever I had a chance to visit. First I thank my friends Yongho Do, Kye-yoon Jang, Myunghoon Kim, Yong-hwan Kwon, Taehyun Yoon, Youngmin Seo, Hanbin Kim, Younghwa Kim, Chadong Kim, Jungsub Lee, Kangwoo Park, Dooyeon Kim, Youngjae Park, Seongpyo Hong, Byungjoo Jeon, Kyung-kyu Lee, and Jaehee Ro. I also want to thank Professor In-Cheol Park, Professor Gyeonghwan Kim, and Professor Tai-kyung Song for providing me with great advices on studying abroad as well as writing me recommendation letters. Special thanks to Professor In-Cheol Park who encouraged me to do Ph.D. in the United States and spared me 2 months of time off during the summer of 2008, so that I can finish my GRE and TOEFL studies.

Last but not least, I express my heartfelt thanks to my family: Hoo-Kyu Rhu, Soon-Ja Kim, Jinsoo Rhu, and Hoili (Haley) Lee. This dissertation would have never made it without the love of my family. I thank my parents for raising me for who I am right now and my brother Jinsoo for taking care of my family while I was studying abroad. Most importantly, I thank my lovely wife Hoili for always being there for me since we first met in 2007. There is no way I could have finished this dissertation without Hoili's love and sacrifice.

Minsoo Rhu

February 2014, Austin, TX

Performance-Efficient Mechanisms for Managing Irregularity in Throughput Processors

Publication No. __________________

Minsoo Rhu, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Mattan Erez

Recent graphics processing units (GPUs) have emerged as a promising platform for general purpose computing and have been shown to be very efficient in executing parallel applications with regular control and memory access behavior. Current GPU architectures primarily adopt the *single-instruction multiple-thread* (SIMT) programming model that balances programmability and hardware efficiency. With SIMT, the programmer writes application code to be executed by scalar threads and each thread is supported with conditional branch and fine-grained load/store instruction for ease of programming. At the same time, the hardware and software collaboratively enable the grouping of scalar threads to be executed in a vectorized *single-instruction multiple-data* (SIMD) in-order pipeline, simplifying hardware design. As GPUs gain momentum in being utilized in various application domains, these *throughput processors* will increasingly demand more efficient execution of irregular applications. Current GPUs, however, suffer from reduced thread-level parallelism,

underutilization of compute resources, inefficient on-chip caching, and waste in off-chip memory bandwidth utilization for highly irregular programs with divergent control and memory accesses.

In this dissertation, I develop techniques that enable simple, robust, and highly effective performance optimizations for SIMT-based throughput processor architectures such that they can better manage irregularity. I first identify that previously suggested optimizations to the divergent control flow problem suffers from the following limitations: 1) serialized execution of diverging paths, 2) lack of robustness across regular/irregular codes, and 3) limited applicability. Based on such observations, I propose and evaluate three novel mechanisms that resolve the aforementioned issues, providing significant performance improvements while minimizing implementation overhead.

In the second half of the dissertation, I observe that conventional coarse-grained memory hierarchy designs do not take into account the massively multi-threaded nature of GPUs, which leads to substantial waste in off-chip memory bandwidth utilization. I design and evaluate a locality-aware memory hierarchy for throughput processors, which retains the advantages of coarse-grained accesses for spatially and temporally local programs while permitting selective fine-grained access to memory. By adaptively adjusting the access granularity, memory bandwidth and energy consumption are reduced for data with low spatial/temporal locality without wasting control overheads or prefetching potential for data with high spatial locality.

Table of Contents

Acknowledgments **v**

Abstract **viii**

List of Tables **xv**

List of Figures **xvi**

Chapter 1. Introduction **1**
- 1.1 Throughput Processors and Irregularity 2
- 1.2 SIMT Challenges upon Irregularity 3
- 1.3 Thesis Statement 5
- 1.4 Contributions 5
- 1.5 Dissertation Organization 8

Chapter 2. Background **10**
- 2.1 The SIMT Programming Model 10
- 2.2 Throughput Processor Architecture 12
- 2.3 System Memory Hierarchy 13
- 2.4 Stack-Based Reconvergence 16

Chapter 3. Improving SIMT Thread-Level Parallelism **20**
- 3.1 Previous SIMT Control-Flow Mechanisms 20
  - 3.1.1 Single-Path Execution Model 21
  - 3.1.2 Dynamic Warp Subdivision 24
  - 3.1.3 Limitation of Previous Models 27
- 3.2 Dual-Path Execution Model 28
  - 3.2.1 Dual-Path Stack Structure 29
  - 3.2.2 Scoreboard 32

3.2.3 Warp Scheduler. . . . 38
3.2.4 Summary of the Benefits of DPE 39
3.3 Evaluation 40
3.3.1 Methodology 40
3.3.2 Benchmarks 41
3.3.3 Results and Analysis 44
3.3.3.1 TLP and SIMD Lane Utilization 44
3.3.3.2 Idle Cycles and Impact on the Memory System 47
3.3.3.3 Overall Performance 49
3.3.3.4 Sensitivity Study 50
3.3.3.5 Implementation and Energy-Efficiency of DPE . 52
3.4 Discussion 54
3.4.1 Path-Forwarding 54
3.4.2 Multi-Path Execution Model 55
3.4.3 DPE for Memory Divergence 56
3.4.4 DPE with a Software-Managed Reconvergence Stack . . 57
3.4.5 Impact on Programmability 57
3.4.6 Alternative to Stack-Based Reconvergence Model 58
3.5 Summary 59

Chapter 4. Enhancing Compute Resource Utilization 61
4.1 Thread-Block Compaction for Irregularity 61
4.1.1 Irregular Control Flow and SIMD Underutilization . . . 62
4.1.2 Thread-Block Compaction 63
4.1.3 TBC Microarchitecture 65
4.1.4 Limitation of TBC 69
4.2 Compaction-Adequacy Prediction 70
4.2.1 Motivation and Key Insights 70
4.2.2 CAPRI Microarchitecture 75
4.2.3 Summary of the Benefits of CAPRI 79
4.3 SIMD Lane Permutation 80
4.3.1 Current Compaction Limitations 80

4.3.2 Aligned Divergence 82
4.3.3 Programmatic Branches and Compactability 85
4.3.4 Motivation and Key Insights 89
4.3.5 Pitfalls of a Random Permutation 92
4.3.6 Balanced Permutation 94
4.3.7 Summary of the Benefits of SLP 97
4.4 Evaluation 98
4.4.1 Methodology 98
4.4.2 CAPRI Results and Analysis 100
4.4.2.1 Prediction Quality 101
4.4.2.2 SIMD Lane Utilization and Idle Cycles 104
4.4.2.3 Overall Performance 106
4.4.2.4 Impact on the Memory System 107
4.4.2.5 Sensitivity Studies 109
4.4.2.6 Implementation and Energy-Efficiency of CAPRI 109
4.4.3 SLP Results and Analysis 110
4.4.3.1 Compactability 111
4.4.3.2 SIMD Lane Utilization 114
4.4.3.3 Overall Performance 116
4.4.3.4 Impact on the Memory System 118
4.4.3.5 Implementation and Energy-Efficiency of SLP . 119
4.5 Discussion 119
4.5.1 Software-Based Permutation 119
4.5.2 Impact on Programmability 120
4.5.3 Cost-Effective Implementation of CAPRI 121
4.5.4 DPE with Compaction 122
4.6 Summary 122

Chapter 5. A Locality-Aware Memory Hierarchy 125
5.1 Conventional Memory System Designs and Data Locality . . . 125
5.1.1 Coarse-Grained Memory Hierarchy 126
5.1.2 Limitation of Coarse-Grained Memory Systems 126

5.2 Designing a Locality-Aware Memory System 130
5.2.1 Fine-Grained Data Management 131
5.2.2 High-Level Overview of LAMAR 133
5.2.3 Bi-modal Granularity Predictor 137
5.2.4 Summary of the Benefits of BGP 142
5.3 Evaluation 143
5.3.1 Methodology 143
5.3.2 LAMAR Results and Analysis 146
5.3.2.1 Prediction Quality, Traffic, and Caching 147
5.3.2.2 Overall Performance 152
5.3.2.3 Impact on DRAM Power Efficiency 152
5.3.2.4 System-Level Power Efficiency 154
5.3.2.5 Sensitivity Study 154
5.3.2.6 Implementation Overhead 159
5.4 Discussion 159
5.4.1 LAMAR on Future Memory Technologies 159
5.4.2 Error Correction 161
5.4.3 Alternative FG Cache Management 161
5.4.4 Other Dynamic Bloom Filter Mechanisms 161
5.4.5 Impact on Programmability 162
5.5 Summary 163

Chapter 6. Future Research Directions 164

6.1 A QoS-Aware Throughput Processor Architecture 164
6.2 Case Study: Priority-Based Cache Allocation 168
6.2.1 Motivation and Key Insights. 168
6.2.2 PCA Implementation. 169
6.2.3 PCA Evaluation and Analysis 172
6.2.3.1 Methodology 172
6.2.3.2 Benchmarks 173
6.2.3.3 TLP Sensitivity 175
6.2.3.4 Overall Performance 176

6.2.3.5 Memory Access Efficiency 177
6.2.4 Discussion and Future Extensions 180
6.3 Summary 182

Chapter 7. Conclusions **183**

Bibliography **186**

Vita **203**

List of Tables

3.1 Simulator configuration for DPE evaluation. 41

3.2 Benchmarks studied for DPE evaluation. 42

4.1 Simulator configuration for CAPRI/SLP evaluation. 98

4.2 Benchmarks studied for CAPRI/SLP evaluation. 101

4.3 CAPRI area overheads 109

5.1 Per thread cache capacity of state-of-the-art CPUs and GPU. 127

5.2 Configuration parameters of *BGP* microarchitecture. 142

5.3 Simulator configuration for LAMAR evaluation. 143

5.4 Benchmarks studied for LAMAR evaluation 146

6.1 Simulator configuration for PCA evaluation. 172

6.2 Benchmarks studied for PCA evaluation 173

List of Figures

2.1 Example thread-hierarchy in CUDA programs 11

2.2 Baseline GPU architecture 12

2.3 SIMT memory access behavior 13

2.4 Baseline coarse-grained memory hierarchy. 15

2.5 High-level overview of the baseline GPU memory hierarchy . . 16

2.6 Example control flow graph 18

3.1 Overview of the single-path execution model 22

3.2 High-level operation of DWS 25

3.3 High-level operation of proposed DPE model 30

3.4 Comparison of scoreboards for SPE and DPE models 32

3.5 Cross-path data dependencies 34

3.6 Managing cross-path data dependencies using DPE scoreboard 35

3.7 Example of (non-)interleavable branches. 43

3.8 Average number of concurrently schedulable paths per warp. . 46

3.9 SIMD lane utilization of SPE/DPE/DWS. 47

3.10 Changes in idle cycles and L1/L2 cache misses 48

3.11 Performance improvements using DPE 50

3.12 Performance sensitivity to cache size and warp scheduler visibility. 51

4.1 Example control flow management with/without compaction . 62

4.2 TBC microarchitecture 66

4.3 Example of thread-block compaction 68

4.4 Example source code containing non-compactable branches . . 70

4.5 Control flows of compaction-ineffective branches 71

4.6 Memory divergence caused by inefficient compaction 73

4.7 Comparison of TBC and CAPRI's execution flow 74

4.8 Example CAPRI execution 76

4.9 Evaluation of compaction-adequacy for a branching point . . . 78

4.10 Average SIMD lane utilization of divergent benchmarks 81

4.11 Example kernel codes containing D-branches. 83

4.12 Example kernel codes containing P-branches. 84

4.13 Active warp status after executing a P-branch 86

4.14 Branch categorization 87

4.15 Compaction rate of P-/D-branches 88

4.16 Example SIMD lane permutation schemes 90

4.17 Code snippet from BACKP that exhibits P-branches 92

4.18 Examples of aligned divergence patterns 93

4.19 Balanced Permutation 95

4.20 Taint analysis for branch categorization 99

4.21 Prediction quality of CAPRI 103

4.22 SIMD lane utilization and (normalized) idle cycle count. . . . 105

4.23 Performance of CAPRI compared to TBC and TBC+ 106

4.24 CAPRI impact on L1/L2 access behavior 107

4.25 CAPRI sensitivity to CAPT size 108

4.26 Compaction rate per branch type with different permutations. 111

4.27 Breakdown of P-branch compaction rate 113

4.28 Breakdown of D-branch compaction rate 115

4.29 Average SIMD lane utilization with SLP 116

4.30 Speedup with SLP 117

5.1 Temporal reuse of cache blocks 128

5.2 Spatial utilization of cache blocks 129

5.3 A memory system with fine-grained data management 133

5.4 High-level overview of LAMAR 134

5.5 The dual bit-array bloom filter microarchitecture 138

5.6 The prediction algorithm of *BGP* 140

5.7 Prediction quality for sectors read into L1 caches 147

5.8 Prediction quality for sectors read into L2 caches 148

5.9 Off-chip byte traffic reduction with LAMAR 148

5.10 Changes in L1 cache miss rates with LAMAR. 150

5.11 Changes in L2 cache miss rates with LAMAR. 151

5.12 Speedup with LAMAR 151

5.13 DRAM power consumption with LAMAR 153

5.14 System power consumption with LAMAR 155

5.15 LAMAR sensitivity study in terms of off-chip traffic 156

5.16 Sensitivity of LAMAR to available TLP 158

6.1 Priority-aware memory scheduler 167

6.2 Temporal reuse of cache blocks with a realistic and ideal cache 174

6.3 Normalized IPC while varying available TLP 175

6.4 Speedup with PCA 176

6.5 Changes in L1/L2 cache miss rates with PCA. 178

6.6 Changes in average L1/L2 cache block reuse with PCA 179

6.7 Changes in off-chip bandwidth utilization with PCA 180

Chapter 1

Introduction

Throughput processors, notably represented by recent graphics processing units (GPUs), have emerged as a promising platform for achieving high performance for various workloads. In order to balance hardware and execution efficiency with programmability and dynamic application control flow, state-of-the-art GPU architectures use the *single-instruction multiple-thread* (SIMT) execution model. The SIMT model allows GPUs to utilize efficient *single-instruction multiple-data* (SIMD) hardware while presenting a simple parallel abstraction to the programmer. With SIMT, the programmer writes codes to be executed by scalar threads but the hardware and software are designed to collaboratively execute groups of scalar threads in vectorized SIMD pipelines (or lanes). Compared to traditional vector/SIMD machines, the SIMT model simplifies mapping parallel applications to SIMD hardware thanks to the native support for per-thread load/store instructions as well as conditional branches. This dissertation focuses on throughput processors based on SIMT-based GPU architectures.

1.1 Throughput Processors and Irregularity

There has been a growing trend of utilizing GPUs as throughput compute engines for high-performance computing (HPC). Thanks to their high computational throughput and memory bandwidth, GPUs have proven effective for exploiting data parallelism in programs operating on large data arrays with well-structured memory accesses [1, 2]. A large body of programs falls in this category of structured control and memory access behavior (e.g., some matrix manipulations [3], bioinformatics [4], computational finance [5], fluid dynamics [6] and others) and their GPU implementations are known to be much faster than their parallel CPU versions; primarily because of these algorithms' regular control flow and memory access patterns [7]. As such, GPUs are widely adopted for accelerating HPC applications, which is highlighted by the fact that in 2013, 54 of the `Top500` list of world's most powerful supercomputers are powered by GPUs [8].

Many application domains, however, are primarily based on algorithms that operate on *irregular* data structures, such as graphs, trees, linked-lists, hash-tables and others. These include applications such as data mining [9], social networks [10], optimization theory [11], n-body simulation [12], and meshing [13]. Due to their complex control flow and irregular memory access patterns, porting these applications to run efficiently on GPUs is much more challenging than regular programs. While recent literature have studied several efficient GPU implementation of such irregular algorithms, the achieved computational efficiency is far from optimal [14, 15, 16]. Graph algorithms,

for instance, are fundamental building blocks for many parallel applications but still perform poorly on GPUs, typically suffering from low SIMD utilization and performance [17, 18, 19]. Given the importance of such applications, efficient handling of program irregularity is vital for the continued success of GPUs as throughput computing platforms.

1.2 SIMT Challenges upon Irregularity

While the SIMT model enhances programmability, the following factors pose challenges in achieving peak performance for highly irregular applications.

Conditional Branch Support. SIMT enables each scalar thread to follow an independent flow of control, despite utilizing vector pipelines for execution. When control flow causes threads within the same group to diverge and take different control flow paths, the underlying SIMD hardware can only partially activate the vector lanes of a single path at a time due to the structural hazard of SIMD. Such hazards are commonly referred to as *control divergence* in the literature, and current GPUs adopt a stack-based reconvergence model (either in hardware or in software) for control flow management. This model partitions the thread group into subgroups that share the same control flow, having only a single subgroup execute at a time while masking out those in different control flows. While the stack model is simple and guarantees correct execution, the serialized execution of basic blocks restricts available thread-level parallelism

and leads to sub-optimal utilization of SIMD lanes for highly irregular control flow, leading to poor performance.

Fine-grained Memory Accesses. The native support for per-thread memory access allows a grouping of scalar threads to exhibit fine-grained *scatter-gather* characteristics. When a group of threads exhibits regular memory access patterns, the cost of memory access is amortized by coalescing them into its minimum number of transactions. For highly irregular applications, however, groups of threads exhibit diverging memory access behavior (e.g., low spatial locality in terms of the memory address each thread is accessing) which causes higher pressure on the memory hierarchy. Such behavior, combined with the massive multithreading nature of GPUs, limits on-chip cache space available within a given timeframe. This exacerbates caching inefficiencies and leads to very low reuse of cache blocks with poor locality. Little research has been conducted on this important problem so far, however, and prior work mainly resorts to a memory hierarchy design that is unaware of these locality characteristics. I observe that this lack of locality consideration in memory systems leads to significant waste in off-chip memory bandwidth utilization and degrades energy-efficiency. The growing interest of accelerating irregular codes, however, necessitates GPUs to manage such memory accesses more effectively.

1.3 Thesis Statement

Future throughput processors will increasingly demand more efficient execution of irregular applications as they gain momentum for general purpose computing. Current throughput processors, however, suffer from reduced thread-level parallelism, underutilization of compute resources, and waste in off-chip bandwidth utilization for highly irregular applications. My thesis demonstrates that future throughput processors can be designed to perform better in executing irregular applications by utilizing thread-level parallelism within/across executing paths, while judiciously coordinating access granularity across the memory hierarchy.

1.4 Contributions

The goal of my thesis is to explore mechanisms that allow GPUs to better manage irreguarity such that they can truly be considered for "general purpose" computing. The first part of this dissertation provides a thorough analysis on the cause of control divergence and identifies that previously suggested optimizations to the divergence problem suffer from the following limitations: 1) serialized execution of diverging paths, 2) lack of robustness across regular/irregular codes, and 3) limited applicability. Based on these observations, I propose and evaluate three novel mechanisms that resolve the aforementioned issues, providing performance-efficient optimizations to GPU control divergence. In the second half of the thesis, I observe that conventional memory hierarchy designs are ill-suited for handling irregular GPU memory

accesses as they sub-optimally utilize on-chip caches as well as off-chip bandwidth. I therefore propose a memory hierarchy design that can efficiently handle irregular memory access patterns in throughput processors such that overall performance as well as energy-efficiency is substantially improved. To summarize the most important contributions of this dissertation:

1. Execution path serialization at divergent branches is a significant challenge of control divergence as it reduces available thread-level parallelism exposed to the thread scheduler. I propose the *dual-path execution* model to alleviate the path serialization problem. The proposed mechanism requires only small changes to the GPU microarchitecture while substantially enhancing thread-level parallelism by allowing the taken and not-taken paths to be executed in an interleaved manner [20]. Unlike previously proposed solutions to the serialization problem [21], the dual-path execution model requires no heuristics, no compiler support, and is robust to changes in architectural parameters.

2. Another significant issue with control divergence is underutilization of compute resources. Recent research has demonstrated that by dynamically *compacting* multiple unmasked threads into a single SIMD group, average compute resource utilization and performance can be improved [22, 23, 24]. Such compaction-based GPU architectures, however, force all candidate threads to synchronize on all conditional branches, effectively generating a hardware-induced compaction-barrier. I observe that such

synchronization overhead is enforced even when an application does not diverge at all, leading to degraded performance even compared to baseline without compaction. To alleviate the unnecessary synchronization overhead, this thesis proposes *compaction adequacy prediction*, which only stalls those threads that are likely to benefit from compaction, while allowing others to bypass compaction and continue executing [25] in order to minimize synchronization.

3. Although prediction of compaction adequacy reduces unnecessary synchronization overhead, the number of applications that practically benefit from compaction is still limited to highly divergent applications. As studied in this thesis, however, for a wide range of applications, there are still unexploited opportunities for compaction. Current GPUs statically assign a *fixed* SIMD lane for each thread based on its thread-ID in a round-robin manner to simplify the structure of the register file. My work demonstrates that such round-robin mapping limits the potential compactability of branching points because control divergence is often exhibited in an aligned/clustered manner on particular SIMD lanes. Based on the insights of the analysis, this work proposes *SIMD lane permutation*, which improves compactability of branches by permuting the home SIMD lanes of threads [26]. While being able to maintain the register file efficiency, SIMD lane permutation substantially reduces the alignement of active threads to lanes and evenly distributes them across all SIMD lanes, significantly improving branch compactability.

4. In addition to the control divergence problem, inefficient utilization of caches and off-chip bandwidth is another key concern for current GPU architectures. The second part of my dissertation shows that irregular memory accesses, combined with massive multithreading, often result in low temporal/spatial reuse of cache blocks because the per-thread cache capacity is small in GPUs. Current GPU memory hierarchies, however, adopt *coarse-grained* memory accesses that always request missed cache blocks in full block-wide granularity. I observe that such coarse-grained memory accesses waste off-chip bandwidth and limit the energy-efficiency of current GPUs for irregular applications by over-fetching unnecessary data. I propose the *locality-aware memory hierarchy* [27] that retains the advantages of coarse-grained accesses for spatially and temporally local programs while permitting selective *fine-grained* access to memory. By adaptively adjusting the access granularity, the memory bandwidth and energy consumption can be substantially reduced for data with low spatial/temporal locality while not wasting control overheads or prefetching potential for data with high spatial locality.

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 reviews necessary background information for understanding contemporary GPU architectures as well as its programming model. Chapter 3 proposes the dual-path execution model as means to enhance thread-level parallelism and

Chapter 4 develops a set of mechanisms to improve SIMD resource utilization. Chapter 5 discusses the limits of conventional memory hierarchy designs and proposes a locality-aware memory hierarchy. Chapter 6 discusses future research directions and Chapter 7 concludes the dissertation.

Chapter 2

Background

This chapter provides basic background information for understanding contemporary GPU architectures and their programming model. We start by briefly reviewing the SIMT programming model in Section 2.1 and the baseline GPU processor architecture in Section 2.2. We then describe the the overall system memory hierarchy in Section 2.3, followed by summarizing the state-of-the-art GPU control flow management schemes in Section 2.4.

2.1 The SIMT Programming Model

The GPU programming model is based on the *single-program multiple-data* concept where a single program (commonly referred to as a *kernel*) is executed by all the threads that the programmer spawns – the programmer writes code to be executed by *scalar* threads, each of which can follow any control flow as well as referencing arbitrary memory addresses. Such execution model is commonly referred to as SIMT and is the most dominant GPU programming and execution model in contemporary GPU architectures (e.g., CUDA [28] and OpenCL [29], supported by GPUs from NVIDIA [30], AMD [31], and Intel [32]).

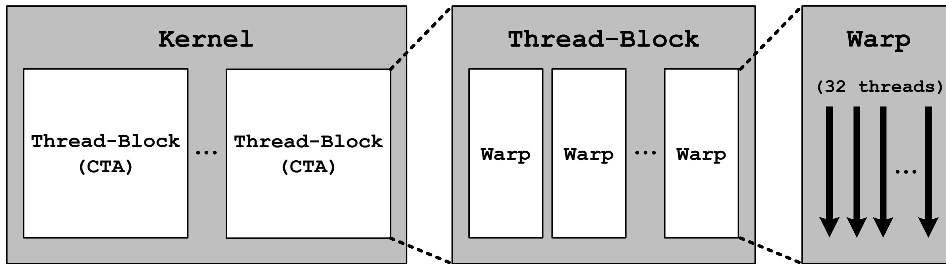


Figure 2.1: Example thread-hierarchy in CUDA programs.

With these programming models, the programmer groups the threads into a *thread-block* (referred to as "cooperative thread arrays" (CTAs) by CUDA and "workgroups" by OpenCL), which is the finest grouping of scalar threads exposed to the programmer. In the actual hardware level, however, each thread-block is decomposed into a finer granularity called *warps* by CUDA and *wavefronts* by OpenCL. In CUDA, for instance, each GPU core schedules the instruction to be executed in a warp of 32 *threads* (wavefront of 64 threads in OpenCL), yet such SIMD grouping of warps is not explicitly exposed to the programmer [28]. The number of threads that constitute a thread-block is an application parameter, and a thread-block typically consists of enough threads to form multiple warps. In general, the SIMT model simplifies mapping parallel application to SIMD hardware thanks to the hardware and software support for collaboratively executing groups of threads in the vectorized SIMD pipelines.

In the rest of this thesis, we will mainly use the terminologies defined in CUDA [28]. Figure 2.1 is an example of a thread hierarchy typically present in CUDA programs.

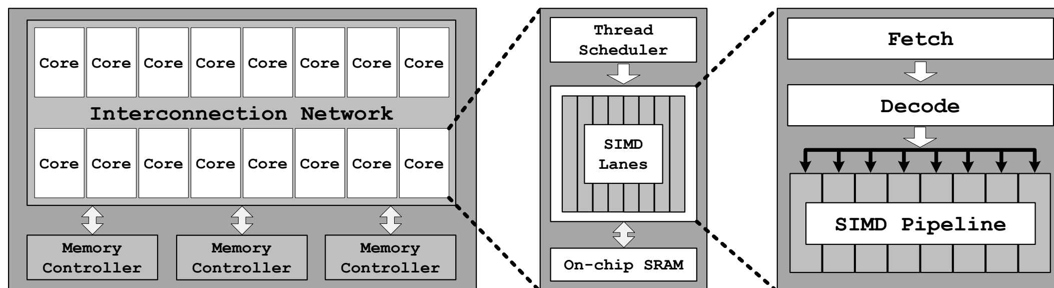
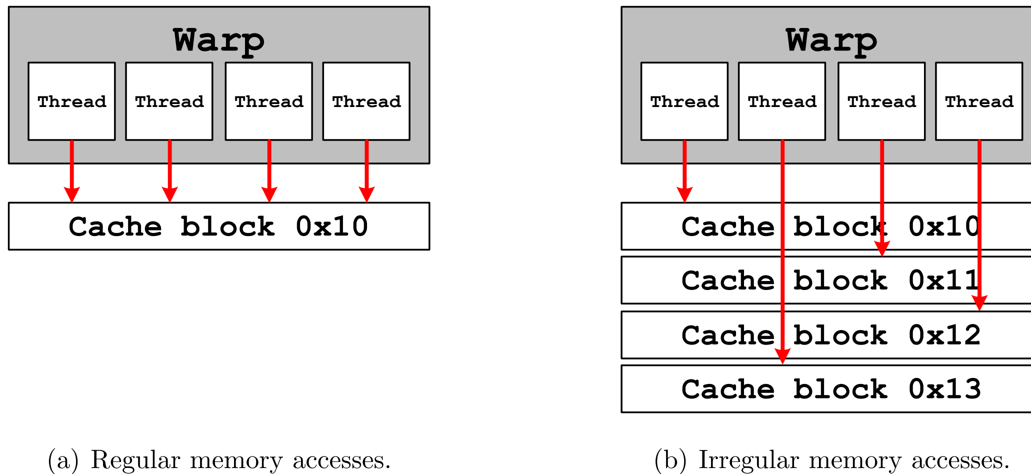


Figure 2.2: High-level overview of the baseline GPU architecture.

2.2 Throughput Processor Architecture

Current GPUs, such as NVIDIA's Tesla [1] or AMD's graphics core next (GCN) [33] architecture, consist of multiple GPU cores (referred to as *streaming multiprocessors* (SMs) by NVIDIA), where each core contains a number of parallel execution lanes that operate in a SIMD fashion: a single instruction is issued to each of the parallel SIMD lanes and that instruction is executed across the lanes simultaneously. Each core also contains an on-chip SRAM array which is divided to be used as an L1 data cache and a local scratchpad.

SIMD lanes are simple in-order pipelines that are optimized for energy-efficiency, hence sophisticated microarchitectural components (e.g., branch predictors, hardware prefetchers, etc ...) that enhance instruction-level parallelism (ILP) do not exist. Rather, GPUs are optimized to leverage thread-level parallelism (TLP) as means to tolerate long-latency operations (e.g., cache misses, transcendental operations, etc ...). Such latency tolerance is achieved by managing a massive number of concurrent threads and allowing them to interleave in a fine-grained manner. Hence, GPU cores contain a large register



(a) Regular memory accesses.

(b) Irregular memory accesses.

Figure 2.3: (Ir)regular memory access behavior and the resulting memory transactions generated by the address coalescer.

file [34] in order to support the large number of concurrent threads, thereby achieving high computational throughput. The number of concurrently executable thread-blocks (and therefore the number of warps/threads) within a GPU core is determined by the number of registers and local scratchpad that each scalar thread requires. The thread scheduler selects one warp per cycle to issue an instruction to all the SIMD lanes within the GPU core. Figure 2.2 is an overview of the baseline processor architecture we assume in this thesis.

2.3 System Memory Hierarchy

SIMT Memory Access Behavior. As discussed in Chapter 1, the SIMT execution model provides fine-grained load/store instructions in a per-thread granularity. This means that a single warp (which consists of 32 scalar threads)

can potentially reference 32 distinct memory regions. In order to save memory access bandwidth, each GPU core contains a hardware address coalescer [28] that compacts spatially local accesses into the minimum number of memory transactions possible. For well-structured, regular memory accesses that are confined within a fixed cache block granularity (Figure 2.3(a)), the hardware coalescer issues a single memory transaction, amortizing the cost of servicing all the scalar threads' request. Warps that exhibit irregular memory access patterns, however, cannot be coalesced and generate multiple cache accesses (Figure 2.3(b))[1]. Such divergent memory accesses (also known as *memory divergence*) cause high cache port contention and put much higher pressure on the overall memory hierarchy, often degrading performance.

Coarse-Grained Memory Hierarchy. GPU manufacturers do not reveal deep microarchitectural details of their memory system, so previous literature [24, 35, 20, 36, 37] assumes a GPU memory hierarchy optimized for *coarse-grained* (*CG*) memory accesses (Figure 2.4). Following memory coalescing, the GPU core requests data (which can be as small as 32 bytes but also up to 128 bytes) from the L1 data cache, where each cache block is sized as 128 bytes. Such 128 bytes cache block design is chosen to accommodate the maximum data request size generated by the GPU core, and a cache miss at L1 will invoke a cache-block wide fill request to the shared L2 cache (which is

[1]The CUDA programming model designates that each warp can invoke one or more 32 to 128 bytes data requests, depending on how many threads are active and how divergent the memory access behavior is.

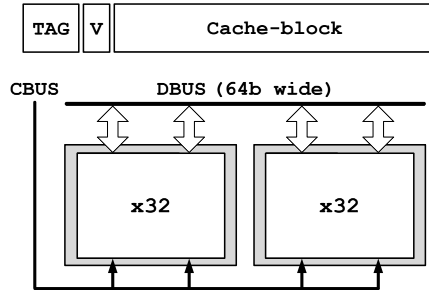


Figure 2.4: Baseline *CG*-only memory hierarchy. `ABUS` designates the control/address bus whereas `DBUS` represents the data bus. Each miss at the cache level invokes a cache block-wide fill request to the lower level of the hierarchy.

the last level cache (LLC) with 128 bytes cache block). If the last level cache misses, the corresponding memory channel will receive the LLC block granularity fill request, and the memory scheduler communicates with the off-chip DRAM to deliver the missed data. Note that the width of the data bus for each memory channel is 64 bits (Figure 2.4), as in Fermi (GF110, [38]), Kepler (GK110, [30]), and SouthernIslands [39], making a 64 bytes minimum-access granularity (8-bursts, 64 bits per burst) with GDDR5 DRAM [40]. To deliver the 128 bytes of data to the LLC, therefore, the memory scheduler issues two consecutive read commands to the off-chip DRAM. Figure 2.5 is a high-level overview of the baseline memory hierarchy, having a uniform interface to the processor with a large (*coarse*) minimum access granularity of cache block size.

While well-structured, regular applications can effectively utilize the baseline *CG* memory system, not all applications can be re-factored to exhibit regular control and memory access patterns. Many emerging applications

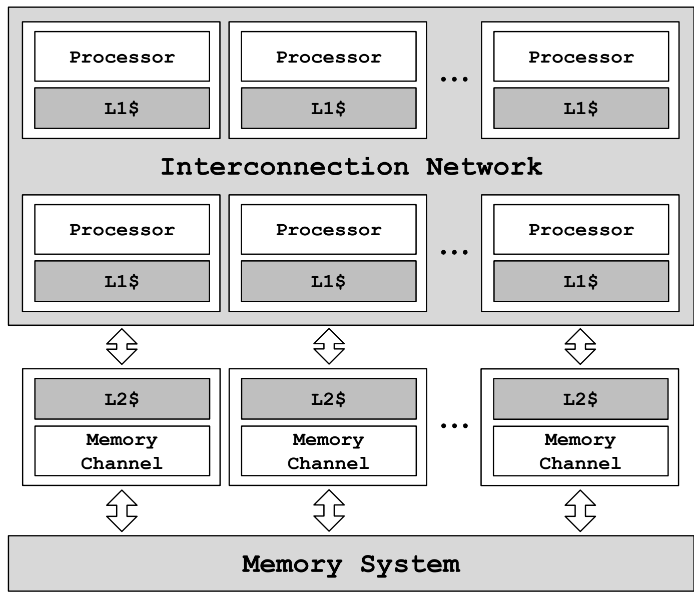


Figure 2.5: High-level overview of the baseline GPU memory hierarchy.

therefore suffer from inefficient utilization of off-chip bandwidth under such baseline configuration and we elaborate on this problem in Chapter 5.

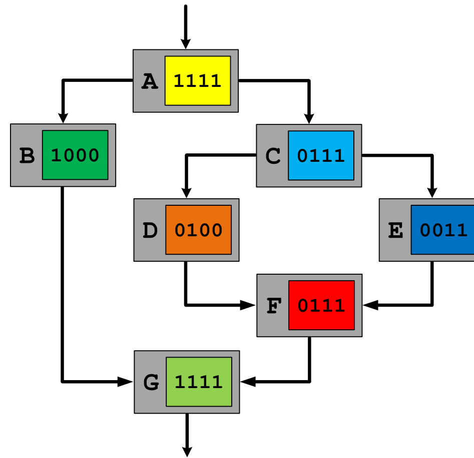
2.4 Stack-Based Reconvergence

While the SIMT execution model enables very efficient hardware, it requires a mechanism to allow each thread to follow its own thread of control, even though only a single uniform operation can be issued across all threads in the same warp. In order to allow independent branching, hardware must provide two mechanisms. The first mechanism ensures that only threads that are on the active path, and therefore share the same program counter (PC), can execute and commit results. This can be done by associating an *active*

mask with each SIMD instruction that executes. Threads that are in the executing SIMD instruction but not on the active control path are masked out and do not commit results. The mask may either be computed dynamically by comparing the explicit PC of each thread with the PC determined for the active path, or alternatively, the mask may be explicitly stored along with information about the control paths. The GPU in Intel's Sandy Bridge [32] stores an explicit PC for each thread while GPUs from AMD and NVIDIA currently associate a mask with each path [30, 31].

The second mechanism determines which single path, of potentially many control paths, is *active* and is executing the current SIMD instruction. The technique for choosing the active path used by current GPUs is *stack-based reconvergence*, which is detailed below.

Control Divergence. A significant challenge with the SIMT model is maintaining high utilization of the SIMD resources when the control flow of different threads within a single warp diverges. There are two main reasons why SIMD utilization decreases with such control divergence (also referred to as branch divergence). The first is that masked operations needlessly consume resources. This problem has been the focus of a number of recent research projects, with the main idea being that threads from multiple warps can be combined to reduce the fraction of masked operations [23, 22, 24, 25]. We further elaborate on this problem in Chapter 4.



- $BR_{X\text{-}Y}$: **Branch instruction that diverges the warp into block *X* and *Y***

Figure 2.6: Example control flow graph. Each warp is assumed to consist of 4 threads and ones and zeros in the control flow graph designate the active and inactive threads in each basic block. The stack-based reconvergence model traverse through the control flow graph in a sequential manner, executing each basic block one at a time (detailed in Section 3.1.1).

The second reason is that execution of concurrent control paths is serialized with every divergence potentially decreasing parallelism. Therefore, care must be taken to allow them to reconverge with one another. In current GPUs, all threads that reach a specific diverged branch reconverge at the *immediate post-dominator* instruction of that branch [41]. The *post-dominator* (PDOM) instruction is the first instruction in the static control flow that is guaranteed to be on both diverged paths [22]. For example, in Figure 2.6, the PDOM of the divergent branch at the end of basic block A (BR_{B-C}) is the instruction that starts basic block G. Similarly, the PDOM of BR_{D-E} at the end of basic block C is the instruction starting basic block F.

Reconverging Diverged Paths using the Stack Model. An elegant way to implement PDOM reconvergence is to treat control flow execution as a stack. Each time control diverges, both the taken and not taken paths are *pushed* onto a stack (in arbitrary order) and the path at the new top of stack is executed. When the control path reaches its reconvergence point, the entry is *popped* off of the stack and execution now follows the alternate direction of the diverging branch. This amounts to a serial depth-first traversal of the control flow graph. Note that only a *single* path is executed at any given time, which is the path that is logically at the top of the stack. There are multiple ways to implement the stack model, including both explicit hardware structures and implicit traversal with software directives [42, 43]. This dissertation assumes the stack model in the context of an explicit hardware approach, which we discuss in-depth in Section 3.1.1. According to prior publications, this hardware approach is used in NVIDIA GPUs [43].

Chapter 3

Improving SIMT Thread-Level Parallelism

This chapter reviews previously studied SIMT control flow management schemes and provides an in-depth analysis on their limitations. Then, the dual-path execution model [20] is introduced as a simple yet highly effective alternative for SIMT control flow. The proposed mechanism only requires small changes to the baseline reconvergence stack model, maintains the same SIMD efficiency, and yet is able to increase available thread-level parallelism and performance.

3.1 Previous SIMT Control-Flow Mechanisms

This section summarizes two representative SIMT control flow management schemes, namely the *single-path execution model* and *dynamic warp subdivision*. We first discuss the single-path execution model which is the SIMT control management scheme widely adopted in contemporary GPU architectures (Section 3.1.1). We then explore dynamic warp subdivision in Section 3.1.2, a recently proposed research project that tackles the execution path serialization issue of the baseline model.

3.1.1 Single-Path Execution Model

Current GPU architectures balance the high efficiency of SIMD execution with programmability and dynamic control. As discussed in Section 2.4, allowing each logical thread to follow independent flow of control under SIMD hardware leads to *control divergence* – when control flow causes threads within the same warp to *diverge* and take different control flow paths. This is currently handled by utilizing a reconvergence predicate stack, which partially serializes execution. The reconvergence stack tracks the program counter (PC) associated with each control flow path, which threads are active at each path (the active mask of the path), and at what PC should a path reconverge (RPC) with its predecessor in the control flow graph [22]. The stack contains the information on the control flow of all threads within a warp, hence each warp has its own stack. Figure 3.1 depicts the per-warp reconvergence stack and its operation on the example control flow shown in Figure 2.6. We describe this example in detail below.

When a warp first starts executing, the stack is initialized with a single entry: the PC points to the first instructions of the kernel (first instruction of block A), the active mask is full, and the RPC (reconvergence PC) is set to the end of the kernel. When a warp executes a conditional branch, the predicate values for both the taken and non-taken paths (left and right paths) are computed. If control diverges with some threads following the taken path and others the non-taken path, the stack is updated to include the newly formed paths (Figure 3.1(b)). First, the PC field of the current *top of the stack*

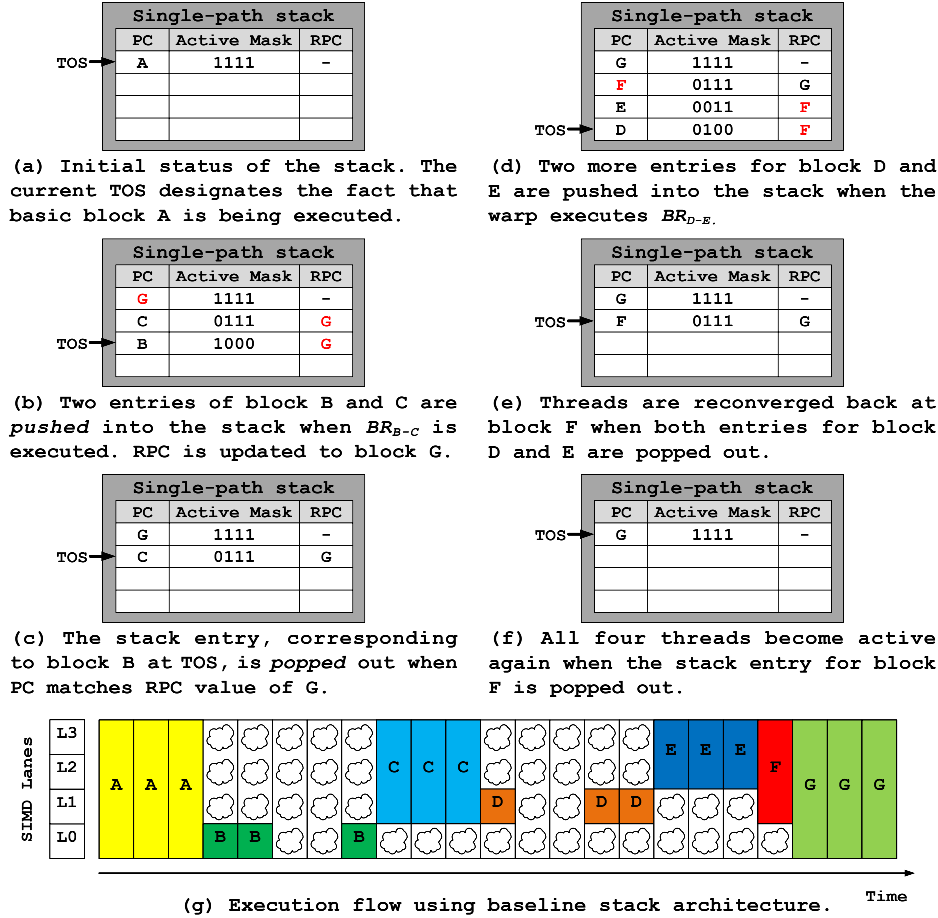


Figure 3.1: High-level operation of the baseline single-path execution model, when executing the control flow graph in Figure 2.6. The ones/zeros inside the active mask field designate the active threads in that block. *Bubbles* in (g) represent idle execution resources (masked lanes or zero ready warps available for scheduling in the GPU core).

(TOS) is modified to the PC value of the reconvergence point, because when execution returns to this path, it would be at the point where the execution reconverges (start of block G in the example). The RPC value is explicitly communicated from software and is computed with a straightforward compiler

analysis [41]. Second, the PC of the right path (block C), the corresponding active mask, and the RPC (block G) is pushed onto the stack. Third, the information on the left path (block B) is similarly pushed onto the stack. Finally, execution moves to the left path, which is now at the TOS. Note that only a *single* path per warp, the one at the TOS, can be scheduled for execution. For this reason we refer to this baseline architecture as the *single-path execution* (SPE) model throughout this thesis.

When the current PC of a warp matches the RPC field value at that warp's TOS, the entry at the TOS is popped off (Figure 3.1(c)). At this point, the new TOS corresponds to the right path of the branch and the warp starts executing block C. As the warp encounters another divergent branch at the end of block C, the stack is once again updated with the left and right paths of blocks D and E (Figure 3.1(d)). Note how the stack elegantly handles the nested branch and how the active masks for the paths through blocks D and E are each a subset of the active mask of block C. When both left and right paths of block D and E finish execution and their corresponding stack entries are popped out, the TOS points to block F and control flow is reconverged back to the path that started at block C (Figure 3.1(e)) – the active mask is set correctly now that the nested branch reconverged. Similarly, when block F finishes execution and the PC equals the reconvergence PC (block G), the stack is again popped out and execution continues along a single path with a full active mask (Figure 3.1(f)).

This example points out the two main deficiencies of the SPE model. First, SIMD utilization decreases every time control flow diverges. SIMD utilization has recently been the focus of active research (e.g., [23, 22, 24, 25]) and this thesis proposes multiple optimizations on this topic in Chapter 4. Second, execution is serialized such that only a single path is followed until it completes and reconverges (Figure 3.1(g)). The SPE model works well for most applications because of the abundant parallelism exposed through multiple warps within thread-blocks. However, for some applications, the restriction of following only a single path does degrade performance. Meng et al. proposed *dynamic warp subdivision* (DWS) [21], which selectively deviates from the reconvergence stack execution model, to overcome the serialization issue. We detail the intuition behind DWS in the next subsection as we use it as a comparison point in our evaluation of the proposed dual-path execution (DPE) model. More recently, Brunie et al. [44] proposed a mechanism that is able to increase intra-warp parallelism, but relies on significant modifications to the underlying GPU architecture described above. Because it targets a different baseline design, we discuss this technique in Section 3.4.6 and qualitatively compare it with DPE.

3.1.2 Dynamic Warp Subdivision

Dynamic warp subdivision (DWS) was proposed to allow warps to interleave the scheduling of instructions from concurrently executable paths [21]. The basic idea of DWS is to treat both the left and right paths of a divergent

Single-path stack (TOS → B)

PC	Active Mask	RPC
G	1111	-
C	0111	G
B	1000	G

Warp-split table

PC	Active Mask	RPC

(a) When BR_{B-C} is executed, the associate post-dominator of block G is examined for eligibility of subdivision: because the number of instructions in block G (which is 3) is larger than the subdivision threshold determined by heuristics, the warp is *not subdivided* and uses the stack to serialize execution of block B and C, rather than using the WST for interleaving the warp-splits. WST remains blank accordingly.

Single-path stack (TOS → C)

PC	Active Mask	RPC
G	1111	-
C	0111	G

Warp-split table

PC	Active Mask	RPC
D	0100	G
E	0011	G

(b) BR_{D-E}, on the other hand, has a post-dominator (block F) smaller than the threshold which allows the warp to be subdivided: the WST is therefore updated with both block D and E, once warp diverges at BR_{D-E}. Note that WST's RPC field is updated to path G (rather than path F which is BR_{D-E}'s post-dominator), which equals the RPC field value at the TOS.

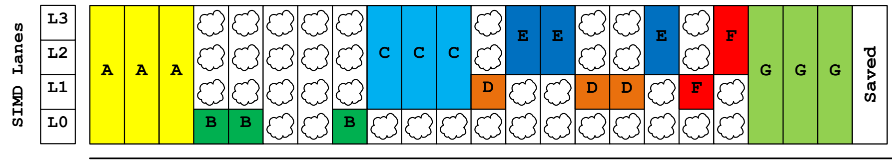
Single-path stack (TOS → G)

PC	Active Mask	RPC
G	1111	-

Warp-split table

PC	Active Mask	RPC
F	0100	G
F	0011	G

(c) Warp-splits registered in WST continue execution until their PC matches its RPC field: compared to SPE which will have these two paths reconverge at block F, DWS allows the two warp-splits to continue execution beyond block F because its RPC field is saved as block G. Once warp-splits arrive path G, the two entries in WST are invalidated, and the reconvergence stack is used to execute path G.



(d) Execution flow using DWS with subdivision threshold of 2.

Figure 3.2: High-level operation of DWS with a subdivision threshold of 2. We assume the same control flow graph in Figure 2.6. Note that all basic blocks are assumed of having exactly three instructions, with the exception of block F that contains only a single instruction.

branch as independently schedulable units, or *warp-splits*, such that diverging path serialization is avoided and *intra*-warp latency tolerance is achieved.[1] With DWS, a divergent branch may either utilize the baseline single-path stack mechanism (Figure 3.2(a)), or instead, ignore the stack and utilize an additional hardware structure, the *warp-split table* (WST), that is used to track the independently-schedulable warp-splits (Figure 3.2(b)). Nested branches within a split warp cause further splits. As with the stack, this successively degrades SIMD efficiency. Unlike the stack, split warps are dynamically and concurrently scheduled and may not reconverge as early as the post-dominator.

To prevent very low SIMD lane utilization, DWS uses a combination of three techniques. First, the WST also contains a reconvergence PC like the stack. This RPC, however, is not the PDOM of the diverging branch, but rather the PDOM of the last entry in the stack. Because the stack cannot be used to track warp-splits, further subdivisions also use this same RPC value and miss many opportunities for reconvergence. This decision increases parallelism and potential latency hiding at the expense of reduced SIMD utilization (the stack could have reconverged nested branches whereas the WST cannot). Second, to reduce the impact of late reconvergence and recursive subdivision, DWS attempts to dynamically and opportunistically recombine warp-splits when two splits happen to reach the same PC. Unlike the PDOM reconver-

[1]Meng et al. [21] also propose to subdivide warps upon memory divergence, which is orthogonal to subdivision at control divergence. My thesis primarily focuses on control divergence subdivision, although we briefly discuss how memory divergence subdivision can be incorporated into the dual-path execution model in Section 3.4.

gence stack mechanism, this opportunistic merging is not guaranteed and may never occur, as illustrated with block F executing twice in Figure 3.2(c–d). Therefore, third, DWS also relies on a heuristic for determining whether to split a warp in the first place: a warp is subdivided only if the divergent branch's immediate post-dominator is followed up by a short basic block of no more than N instructions. Meng et al. suggest that this *subdivision threshold* (N) should be 50 instructions [21], which we refer to as DWS_{50}.

As I demonstrate in my thesis, DWS cannot consistently balance increased parallelism and SIMD utilization and often degrades performance when compared to the baseline SPE. The threshold heuristic is sensitive, with small values of N often preventing splits and not increasing thread-level parallelism (TLP) significantly, while high values of N split too aggressively and exhibit low SIMD utilization. Furthermore, the WST adds complexity and the compiler may need to change heuristics based on the specific parameters of the hardware and application. In contrast, dual-path execution is very robust; it does not degrade performance compared to the baseline and outperforms DWS in all but one experiment despite DWS exposing greater parallelism.

3.1.3 Limitation of Previous Models

As discussed in previous subsections, both SPE and DWS are able to address only one aspect of the control divergence issue while overlooking the other. SPE uses simple hardware and an elegant execution model to maximize SIMD utilization with structured control flow, but *always* serializes execution

with only a single path schedulable at any given time. DWS can interleave the scheduling of multiple paths and increase TLP, but this sacrifices SIMD lane utilization. The *dual-path execution* (DPE) model, on the other hand, always matches the utilization and SIMD efficiency of the baseline SPE while still enhancing TLP in some cases. DPE keeps the elegant reconvergence stack model and the hardware requires only small modifications to utilize up to two interleaved paths. The following section describes the microarchitectural aspects of DPE, followed by a detailed evaluation and discussion.

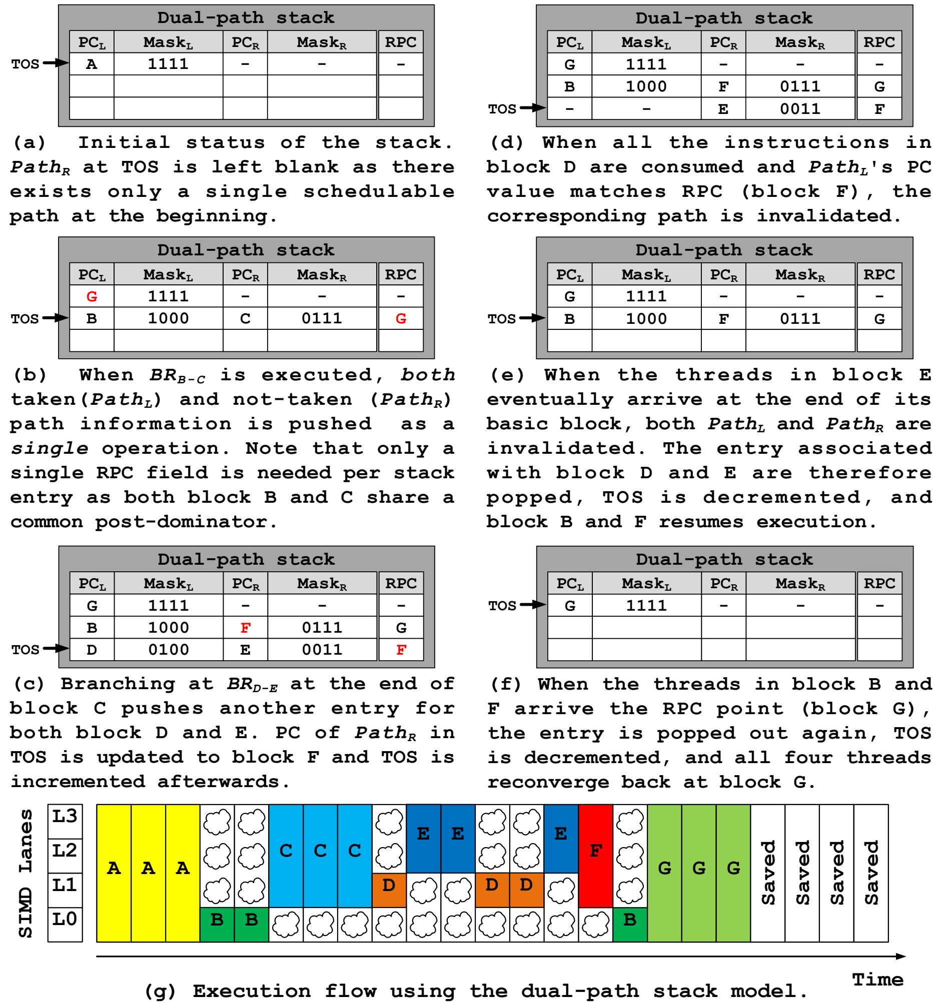
3.2 Dual-Path Execution Model

This work is motivated by the key observation that previous architectures either rely on stack-based reconvergence and restrict parallelism to a single path at any given time, or that stack-based reconvergence is abandoned leading to much more complex implementations and possible degradation of SIMD efficiency (with DWS). My approach maintains the simplicity and effectiveness of stack-based reconvergence but exposes greater parallelism to the scheduler. With DPE, the execution of up to two separate paths can be interleaved, while reconvergence is identical to the baseline stack-based reconvergence. Support for DPE is only required in a small number of components within the GPU microarchitecture and requires no support from software. Specifically, the stack itself is enhanced to provide up to two concurrent paths for execution, the scoreboard is modified to track dependencies of two concurrent paths and to correctly handle divergence and reconvergence, and the warp

scheduler is extended to handle up to two schedulable objects per warp. The details of the DPE microarchitecture is explained below and we use a running example of the control flow in Figure 2.6 corresponding to the code shown in Figure 3.5.

3.2.1 Dual-Path Stack Structure

DPE extends the hardware stack used in many current GPUs to support two concurrent paths of execution. The idea is that instead of pushing the taken and fall-through paths onto the stack one after the other, in effect serializing their execution, the two paths are maintained in parallel. A stack entry of the dual-path stack architecture thus consists of three data elements: a) PC and active mask value of the left path ($Path_L$), b) PC and active mask value of the right path ($Path_R$), and c) the RPC (reconvergence PC) of the two paths. We use the generic names left and right because there is no reason to restrict the mapping of taken and non-taken paths to the fields of the stack. Note that there is no need to duplicate the RPC field within an entry because $Path_L$ and $Path_R$ of a divergent branch have a common reconvergence point. Besides the data fields that constitute a stack entry, the other components of the control flow hardware, such as the logic for computing active masks and managing the stack, are virtually identical to those used in the baseline stack architecture. The dual-path stack architecture exposes the two paths for execution on a divergent branch and can improve performance when this added



	PC_L	$Mask_L$	PC_R	$Mask_R$	RPC
TOS →	A	1111	-	-	-

(a) Initial status of the stack. $Path_R$ at TOS is left blank as there exists only a single schedulable path at the beginning.

	PC_L	$Mask_L$	PC_R	$Mask_R$	RPC
	G	1111	-	-	-
TOS →	B	1000	C	0111	G

(b) When BR_{B-C} is executed, *both* taken($Path_L$) and not-taken ($Path_R$) path information is pushed as a *single* operation. Note that only a single RPC field is needed per stack entry as both block B and C share a common post-dominator.

	PC_L	$Mask_L$	PC_R	$Mask_R$	RPC
	G	1111	-	-	-
	B	1000	F	0111	G
TOS →	D	0100	E	0011	F

(c) Branching at BR_{D-E} at the end of block C pushes another entry for both block D and E. PC of $Path_R$ in TOS is updated to block F and TOS is incremented afterwards.

	PC_L	$Mask_L$	PC_R	$Mask_R$	RPC
	G	1111	-	-	-
	B	1000	F	0111	G
TOS →	-	-	E	0011	F

(d) When all the instructions in block D are consumed and $Path_L$'s PC value matches RPC (block F), the corresponding path is invalidated.

	PC_L	$Mask_L$	PC_R	$Mask_R$	RPC
	G	1111	-	-	-
TOS →	B	1000	F	0111	G

(e) When the threads in block E eventually arrive at the end of its basic block, both $Path_L$ and $Path_R$ are invalidated. The entry associated with block D and E are therefore popped, TOS is decremented, and block B and F resumes execution.

	PC_L	$Mask_L$	PC_R	$Mask_R$	RPC
TOS →	G	1111	-	-	-

(f) When the threads in block B and F arrive the RPC point (block G), the entry is popped out again, TOS is decremented, and all four threads reconverge back at block G.

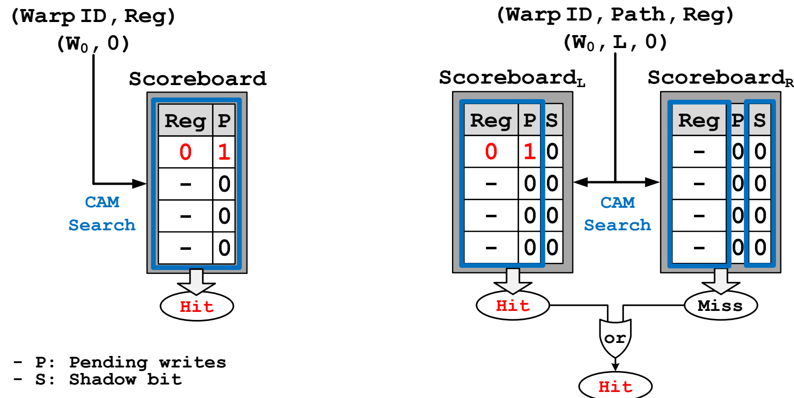
(g) Execution flow using the dual-path stack model.

Figure 3.3: High-level operation of dual-path execution model. Figure assumes the same control flow graph and assumptions in Figure 3.2.

parallelism is necessary. Figure 3.3 illustrates the high-level operation of DPE, which is described below for the cases of divergence and reconvergence.

Handling Divergence. A warp starts executing on one of the paths, for example the left path, with a full active mask, the PC set to the first instruction in the kernel and the RPC set to the last instruction ($Path_L$ in Figure 3.3(a)). The warp then executes in identical way to the baseline single-path stack until a divergent branch executes. When the warp executes a divergent branch, the dual-path stack architecture pushes a single entry onto the stack, which represents both sides of the branch, rather than pushing two distinct entries as done with the baseline SPE. The PC field of the block that diverged is set to the RPC of both the left and right paths (block G in Figure 3.3(b)), because this is the instruction that should execute when control returns to this path. Then, the active mask and PC of $Path_L$, as well as the same information for $Path_R$ are pushed onto the stack, along with their common RPC and updating the TOS (Figure 3.3(b)). Because it contains the information for both paths, the single TOS entry enables the warp scheduler to interleave the scheduling of active threads at both paths as depicted in Figure 3.3(g). If both paths are active at the time of divergence, the one to diverge (block C in Figure 3.3(b)) first pushes an entry onto the stack, and in effect, suspends the other path (block B in Figure 3.3(c)) until control returns to this stack entry (Figure 3.3(e)). Note that the runtime information required to update the stack entries is exactly the same as in the baseline single-path stack model.

Handling Reconvergence. When either one of the basic blocks at the TOS arrives at the reconvergence point and its PC matches the RPC, the block is



(a) Input register number is compared in parallel with all the scoreboard entries' (Reg:P) field for a match.

(b) Dependency is determined by OR-ing own path's (Reg:P) match and the other path's (Reg:S) match.

Figure 3.4: Comparison of scoreboards used for single-path and dual-path execution model. Each scoreboard reflects the status after executing the first *load* instruction in path A of Figure 3.5

invalidated (block D in Figure 3.3(d)). Because the right path is still active, though, the entry is not yet popped off of the stack. Once both paths arrive at the RPC, the stack is popped and control is returned to the next stack entry (Figure 3.3(e–f)).

3.2.2 Scoreboard

Recent GPUs from NVIDIA, such as Fermi [38], allow threads within the same warp to be issued back to back using a per warp scoreboard to track data dependencies. One possible implementation of the scoreboard ([45]) is a content-addressable-memory (CAM) structure that is indexed with a register number and a warp ID and returns whether that register is pending write-back

for that warp (Figure 3.4(a)). When an instruction is decoded, the source and destination registers are searched in the scoreboard and only instructions with no RAW/WAW hazards are considered for scheduling. Once an instruction is scheduled for execution, the scoreboard is updated to show the instruction's destination register as pending. When the register is written back, the scoreboard is updated and the pending bit is cleared. To support multiple concurrent paths per warp, the scoreboard must be modified to track the register status of both the right and left paths of each warp independently while still correctly handling divergence and reconvergence when dependencies are crossed from one path to the other.

DPE accomplishes this with two modifications to the scoreboard. First, we extend the scoreboard to track the left and right path separately (Figure 3.4(b)). This, in essence, doubles the scoreboard so that the two paths can execute concurrently with no interference. Second, we add a *shadow* bit to each scoreboard entry, which is used to ensure correct execution when diverging and reconverging. To explain how the shadow bits are used, we first introduce the issues with divergence and reconvergence. There are four scenarios that must be considered (Figure 3.5):

1. Unresolved pending writes before divergence (e.g., $r0$ on path A) should be visible to the other path (e.g., $r0$ on path C) after divergence, and further, both paths need to know when $r0$ is written back. Ignoring either aspect will lead to either incorrect execution or deadlock. If a register

Example code

```
// Path A
load r0, MEM[~];

if( ){ // Path B
   load r1, MEM[~];
}
else{ // Path C
   add r5, r0, r2;
   …
   if( ){ // Path D
   add r4, r1, r3;
   }
   else{ // Path E
   sub r4, r1, r3;
   }
   // Path F
   …
   load r7, MEM[~];
}
// Path G
add r8, r1, r7;
```

Left Path	Right Path
`// Path A` `load r0, MEM[~];`	
Divergence	

`if( ){ // Path B`
`load r1, MEM[~];`
`}`
	`else{ // Path C`
	`add r5, r0, r2;`
	`…`

Divergence	

`if( ){ // Path D`
`add r4, r1, r3;`
`}`
	`else{ // Path E`
	`sub r4, r1, r3;`
	`}`

Reconvergence	

	`// Path F`
	`…`
	`load r7, MEM[~];`
	`}`

Reconvergence	
`// Path G` `add r8, r1, r7;`	

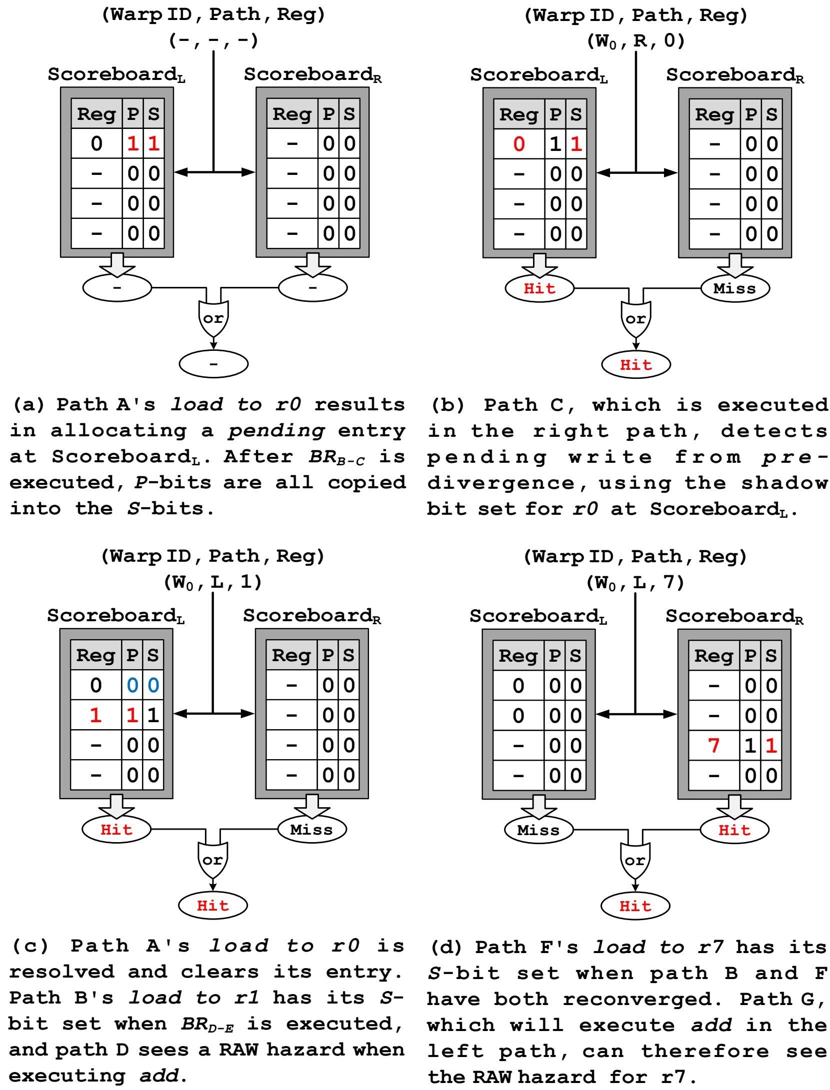
- Code segments placed horizontally are paths scheduled simultaneously.
- Code segments are placed vertically in execution order.

Figure 3.5: Data dependencies across different execution paths (control flow of Figure 2.6)

that is not yet ready is used, an incorrect result may be generated. Conversely, if a register is assumed pending and is never marked as ready, execution will deadlock.

2. Unresolved pending writes before reconvergence (e.g., $r7$ on path F) should be visible to the other path ($r7$ at path G) after reconvergence.

3. If a register number is the destination register of an instruction past a divergence point, then this register should not be confused with the same register number on the other path. Treating this as a false dependency



(a) Path A's *load to r0* results in allocating a *pending* entry at $Scoreboard_L$. After BR_{B-C} is executed, *P*-bits are all copied into the *S*-bits.

(b) Path C, which is executed in the right path, detects pending write from pre-divergence, using the shadow bit set for *r0* at $Scoreboard_L$.

(c) Path A's *load to r0* is resolved and clears its entry. Path B's *load to r1* has its *S*-bit set when BR_{D-E} is executed, and path D sees a RAW hazard when executing *add*.

(d) Path F's *load to r7* has its *S*-bit set when path B and F have both reconverged. Path G, which will execute *add* in the left path, can therefore see the RAW hazard for r7.

Figure 3.6: Example of how the proposed scoreboard handles data dependencies across different paths in Figure 3.5.

between the paths may hurt performance but does not violate correctness (e.g., $r1$ on path B is a different register than $r1$ on paths D/E).

4. Similarly to the case above, if the register number on two different paths is a destination in both paths concurrently, then writes to this register number from the two paths should not be confused with one another. Thus, enforcing a false dependency is a poor design option because it will lead to the two paths being serialized as one path waits to issue until after the other path writes back.

Maintaining separate left and right scoreboards addresses the fourth scenario listed above and allows two independent paths to operate in parallel, but on its own cannot handle cross-path dependencies resulting from divergence and reconvergence. The proposed scoreboard design handles the first three cases conservatively by treating a pending write from before a divergence or reconvergence as pending on both paths after divergence/reconvergence, regardless of which path it originated in. This guarantees that no true dependency will be violated. To achieve this behavior, when a path diverges, the pending bits of its scoreboard are copied to its shadow bits. When querying the scoreboard for a register in one path, the shadow bits in the other path are also examined. If either the path's scoreboard or the shadow in the other scoreboard indicate a pending write, the path stalls (Figure 3.4(b)). In our example, this mechanism ensures that path C correctly waits for the pending write of $r0$ from path A (Figure 3.6(a–b)). Upon a writeback, both the shadow and pending bits of the original scoreboard of the instruction are cleared, freeing instructions on both paths to schedule (Figure 3.6(c)). This requires propagating a single additional bit down the pipeline to indicate whether a

writeback is to the left or right scoreboards. A similar procedure is followed for reconvergence to guarantee correct cross-path dependencies, as shown with the dependency on $r7$ from path F to path G (Figure 3.6(d)).

At the same time, our design does not create dependencies between concurrent left and right paths. For example, after the divergence of BR_{D-E}, the shadow bits for $r4$ are not set, and thus, $r4$ is tracked independently in the left and right scoreboards. While this mechanism ensures correct execution and avoids serialization as described above, it may introduce false dependencies that partially stall execution. For example, the write to $r1$ on path B is unrelated to the reads of $r1$ on paths C and D. The shadow bit for $r1$ on the left scoreboard is set when the paths diverge at BR_{D-E}, which unnecessarily stalls the execution of blocks D and E. On the other hand, this false dependency also ensures that $r1$ generated in path B is written back before the dependent instruction in path G executes (Figure 3.6(d)).

While a much more sophisticated scoreboard structure that can filter out such false dependencies can be designed, our experiments indicate it will provide little benefit of a maximum 1% performance improvement (Section 3.3.3.4). The cost of a non-conservative scoreboard, on the other hand, would be high because it would require more information to propagate in the pipeline and additional logic to decide when and when not to wait. The proposed scoreboard is simple in both design and operation.

3.2.3 Warp Scheduler.

In the baseline SPE architecture, the warp scheduler chooses which of the ready-to-execute warps should be issued in the next cycle. Because of the large number of warps and execution units, some GPUs utilize multiple parallel schedulers with each scheduler responsible for a fixed subset of the total warps and also a subset of the execution lanes [38, 30]. For example, NVIDIA's Fermi GPU has two warp schedulers within each GPU core; one scheduler for even-numbered warps and the other for odd-numbered warps with each scheduler responsible for scheduling an instruction for 16 of the 32 lanes within the core [38]. DPE can expose up to twice the number of schedulable units as each warp can have both a left and a right path at the same time. This thesis assumes that the scheduler can be extended to support this greater parallelism by simply doubling the number of entries. Because each warp has two entries, a single additional selection level to choose which of the two entries competes with other ready warps is all that is required from the logic perspective.

In addition to this expanded scheduler that has twice as many schedulable entries, we also experiment with a constrained warp scheduler that maintains the same number of entries as SPE. In this constrained configuration, each warp is allocated a single entry and only one path, which is determined in the previous cycle, can be considered for scheduling at any time. In order to not lose scheduling opportunities when only one path is available, or when only one path is ready, we do not alternate between the paths on every cy-

cle. Instead, we rotate which path will be available for scheduling whenever the current schedulable path encounters a cache miss or executes another long latency operation (e.g. a transcendental function).

3.2.4 Summary of the Benefits of DPE

As described above, the dual-path execution model extends current GPU designs with greater parallelism at very low cost. It requires no change to the underlying execution model and does not sacrifice SIMD efficiency. The extension to the stack itself is simple and only requires small modifications to existing structures. The warp scheduler also requires only a straightforward extension to support the greater level of parallelism exposed. The most significant change is to the scoreboard, and we show how to extend the baseline scoreboard to support two paths in a cost-effective manner. While the proposed solution amounts to replicating the scoreboard structure, it does not add significant complexity because the left and right scoreboard do not directly interact and the critical path of the dependency-tracking mechanism is only extended by a multiplexer to select the pending or shadow bits and the OR-gate shown in Figure 3.4(b). Section 3.3.3.5 provides a qualitative discussion on the implementation overheads of DPE and its energy-efficiency.

In the following sections we demonstrate the advantages of the extra parallelism over single-path execution, as well as added robustness and performance compared to DWS.

3.3 Evaluation

This section describes the evaluation methodology, followed by a detailed evaluation of the DPE model. We explore DPE's impact on TLP, resource utilization, number of idle cycles, overall performance, sensitivity to key parameters, and implementation overheads. All average values are based on harmonic means.

3.3.1 Methodology

We model the microarchitectural components of dual-path execution using GPGPU-Sim [46, 47], which is a detailed cycle-based performance simulator of a general purpose GPU architecture supporting CUDA version 3.1 and its PTX ISA. In addition to the baseline scoreboard provided as a default with GPGPU-Sim, we model the conservative scoreboard and the warp scheduler that can schedule both the left and right paths arbitrarily. We also implemented an optimistic scoreboard that does not add any false dependencies and a constrained warp scheduler that alternates between the left and right path of each warp (all four mechanisms described in Section 3.2). DWS with PC-based reconvergence has been implemented and simulated as described in Section 3.1.2 and by Meng et al. [21]. We do not constrain DWS resources and model its warp scheduler and scoreboard as perfect; i.e., there are enough scoreboard resources to track an arbitrary number of warp splits, no false dependencies are introduced, and any number of warp splits can be scheduled together with no restriction. Because DWS is sensitive to the heuristic guiding

Table 3.1: Simulator configuration for DPE evaluation.

Number of GPU cores	15
Threads per GPU core	1536
Threads per warp	32
SIMD lane width	32
Registers per GPU core	32768
Shared memory per GPU core	48KB
Number of warp schedulers	2
Warp scheduling policy	Oldest warp first [35]
L1 cache (size/associativity/block size)	16KB/4-way/128B
L2 cache (size/associativity/block size)	768KB/8-way/128B
Memory bandwidth	177.6 GB/s
Memory controller	Out-of-order (FR-FCFS)

subdivision, we simulated DWS with a range of subdivision threshold values. In general, the simulator is configured to be similar to NVIDIA's Fermi architecture using the configuration file provided with GPGPU-Sim [48]. The key parameters used are summarized in Table 3.1 and we explicitly mention when deviating from these parameters for sensitivity analysis.

3.3.2 Benchmarks

DPE has been studied with 27 benchmarks from Rodinia [14], Parboil [49], CUDA-SDK [3], the benchmarks provided with GPGPU-Sim [47], a number of applications from CUDA Zone [2] that can be simulated with GPGPU-Sim, and MCML [50]. The benchmarks studied are ones whose kernel can execute to completion on GPGPU-Sim. We report the results of the first 5 iterations of the kernel of MCML (each iteration results in instructions per cycle (IPC) with near zero variation among different iterations) due to its long simulation time. Note that this section summarizes the detailed results

Table 3.2: Benchmarks studied for DPE evaluation.

Interleavable			
Abbreviation	**Description**	**#Instr.**	**Ref.**
LUD	LU Decomposition	39M	[14]
QSort	Quick Sort	60M	[2]
Stencil	3D Stencil Operation	115M	[49]
RAY	Ray Tracing	250M	[47]
LPS	Laplace Solver	72M	[47]
MUMpp	MUMmerGPU++	148M	[51]
MCML	Monte Carlo for ML Media	303B	[50]
Non-interleavable			
Abbreviation	**Description**	**#Instr.**	**Ref.**
DXTC	DXT Compression	18B	[3]
BFS	Breadth-First Search	16M	[47]
PathFind	Path Finder	639M	[14]
NW	Needleman-Wunsch	51M	[14]
HOTSPOT	Hot-Spot	110M	[14]
BFS2	Breadth-First Search 2	26M	[14]
BACKP	Back Propagation	190M	[14]

for the 14 benchmarks shown in Table 3.2, because other benchmarks execute in an identical way with SPE, DPE, and DWS, as represented by DXTC and BACKP. The reason for the identical behavior is that the structure of the control flow in these kernels does not expose any added parallelism with DPE and that the heuristic that guides DWS always results in no warp splits. Within the 14 benchmarks we discuss, half do not benefit from DPE because the branch structure does not result in distinct left and right paths that can be interleaved (categorized as non-interleavable in Table 3.2). We discuss this further in the next section. Note that these benchmarks are impacted by DWS and we evaluate their behavior with DWS.

Example Code) Branch with only the true path active on divergence.

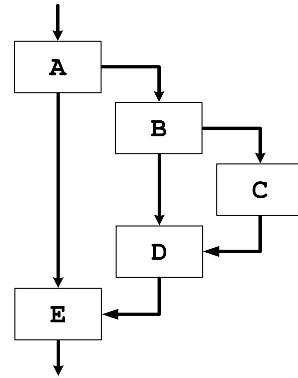
< Code snippet from the kernel of ***BFS*** benchmark >

```
int tid = blockIdx.x*MAX_THREADS_PER_BLOCK + threadIdx.x;

// Block A
if( tid<no_of_nodes && g_graph_mask[tid] )    // BR_B-E
{
   // Block B
   …

   // End of Path B
   if(!g_graph_visited[id])  // BR_C-D
   {
   // Block C
      …
   }
   // Block D
}
// Block E
```

< Corresponding control flow graph >



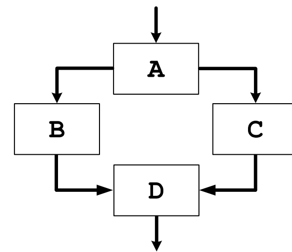
(a) Non-interleavable branches.

Example Code) Branch with both true/false path active on divergence.

< Code snippet from the kernel of ***LUD*** benchmark >

```
// Block A
if(threadIdx.x < BLOCK_SIZE){    // BR_B-C
   // Block B
   idx = threadIdx.x;
   array_offset = offset*matrix_dim+offset;
   for (i=0; i < BLOCK_SIZE/2; i++){ … }
   …
}
else{
   // Block C
   idx = threadIdx.x-BLOCK_SIZE;
   array_offset =(offset+BLOCK_SIZE/2)*matrix_dim+offset;
   for (i=BLOCK_SIZE/2; i < BLOCK_SIZE; i++){ … }
   …
}
// Block D
```

< Corresponding control flow graph >



(b) Interleavable branches.

Figure 3.7: Example of (non-)interleavable branches.

3.3.3 Results and Analysis

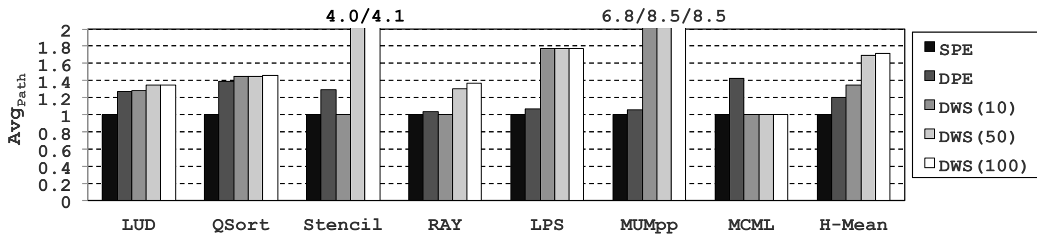
3.3.3.1 TLP and SIMD Lane Utilization

The goal of DPE is to increase the parallelism available to the warp scheduler by allowing both the taken and non-taken paths of a branch to be interleaved. Not all divergent branches, however, are *interleavable* because many branches have only an *if* clause with no *else*. With such branches, the reconvergence point and the fall-through point are the same and the non-taken path is empty. BFS, for instance, is known to be highly irregular (average SIMD lane utilization of only 32%) with significant portion of its branches diverging. All its divergent branches, however, are ones with only *if* and no *else* clause (Figure 3.7(a)), which leads to all threads in $Path_R$ arriving at the reconvergence point immediately: threads branching into block E at BR_{B-E} and ones branching into block D at BR_{C-D} all have their next PC equal to RPC upon divergence and are deactivated until threads in the other path reconverge. We refer to such divergent branches as *non-interleavable*, and to branches that result in both non-empty left and right paths as *interleavable* (Figure 3.7). A benchmark often contains a mix of interleavable and non-interleavable branches, and only the interleavable ones have potential for interleaving with DPE. I therefore define Avg_{Path} (Equation 3.1) to quantify each benchmark's potential for interleaving, where N is the *total number of warp instructions issued throughout the execution of the kernel* and $NumPath_i$ is the *total number of concurrently schedulable paths available at the top of the stack* when the *i-th* warp is issued.

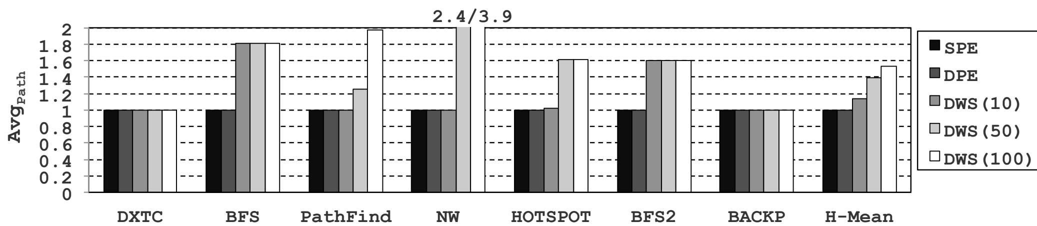
$$Avg_{Path} = \frac{\sum_{i=1}^{N} NumPath_i}{N} \tag{3.1}$$

SPE, which can only schedule the single path at the TOS, always has $NumPath_i$ equal to 1, and hence, has $Avg_{path} = 1$ for all benchmarks. DPE, on the other hand, has $Avg_{path} > 1$ for interleavable benchmarks, as $NumPath_i$ equals 2 when an interleavable branch, which generates both $Path_L$ and $Path_R$ at the TOS, is executed. Note that Avg_{Path} is 1 with DPE as well when all the divergent branches within the benchmark are non-interleavable. When DWS is used, $NumPath_i$ equals 1 when the warp is scheduled in non-subdivided mode as it uses the conventional stack to serialize execution. When a warp is subdivided, however, $NumPath_i$ is equal to the total number of valid entries (hence the number of valid warp-splits) in the WST. Accordingly, non-interleavable benchmarks *can* have an Avg_{Path} value larger than 1 when DWS is used.

Figure 3.8 shows Avg_{Path} for all 14 benchmarks with three different subdivision thresholds, with DWS_{10} being the most conservative about subdividing warps and DWS_{100} the most aggressive. Overall, both DPE and DWS are able to achieve significant increases in Avg_{Path} value across the interleavable benchmarks (an average increase of 20% and 71% for DPE and DWS_{100}, respectively), thereby exposing more TLP for the warp scheduler. DWS_{100} and DWS_{50} expose significantly more TLP than DPE and also increase TLP for non-interleavable benchmarks. As discussed in Section 3.2, the improvement in TLP with DWS comes at the cost of decreased SIMD utilization. Figure 3.9 summarizes the average SIMD lane utilization achieved across these



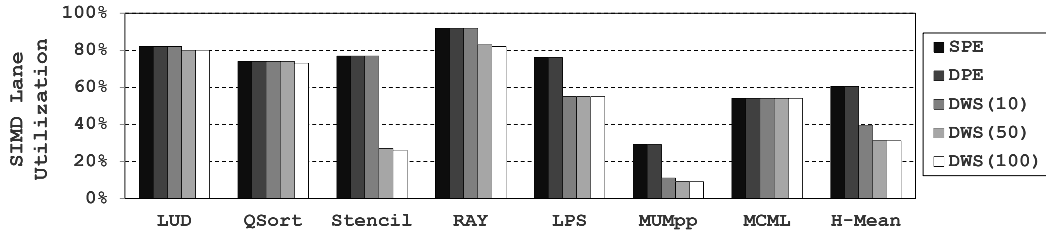
(a) Interleavable benchmarks.



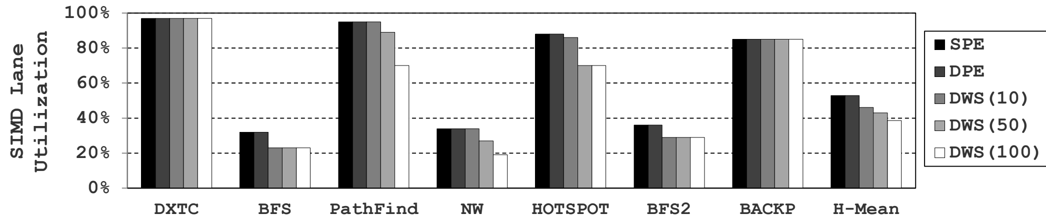
(b) Non-interleavable benchmarks.

Figure 3.8: Average number of concurrently schedulable paths per warp.

benchmarks. While, as expected, DPE shows no loss in SIMD lane utilization across all benchmarks, DWS sacrifices a large fraction of SIMD utilization in many cases. With the exception of LUD, QSort, MCML, DXTC, and BACKP, DWS_{50} and DWS_{100} reduce SIMD efficiency by a large amount for all benchmarks: an average 48.1%/48.5% loss for interleavable benchmarks and 18.6%/27.1% loss for non-interleavable ones, respectively. This implies that subdivision was performed far too aggressively, sacrificing efficiency for increased TLP. In other words, DWS_{50} and DWS_{100}'s high Avg_{Path} is obtained at the cost of having basic blocks that would have been executed once (when using the reconvergence stack of SPE or DPE) to instead execute as many warp-splits. DWS_{10} loses less SIMD efficiency because it splits fewer warps,



(a) Interleavable benchmarks.



(b) Non-interleavable benchmarks.

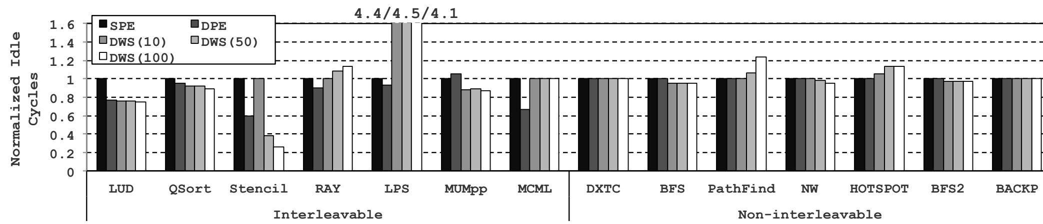
Figure 3.9: SIMD lane utilization of SPE/DPE/DWS.

but SIMD utilization is still significantly decreased (24.6% on average overall) and the conservative heuristic fails to improve TLP in some cases.

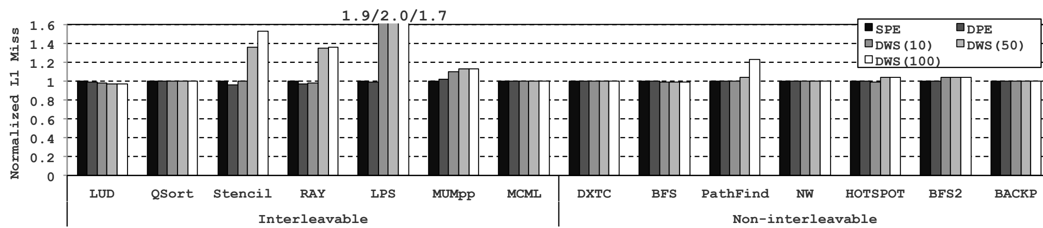
3.3.3.2 Idle Cycles and Impact on the Memory System

Figure 3.10 illustrates the impact the different schemes have on the number of idle cycles and L1/L2 cache misses. Overall, DPE can reduce the number of idle cycles by an average of 11% for interleavable benchmarks while matching SPE with non-interleavable ones. The only exception is MUMpp where the interleaving of diverging paths disrupts the access pattern to the L1 cache and increases the number of misses by 2% and idle cycles by 4%.

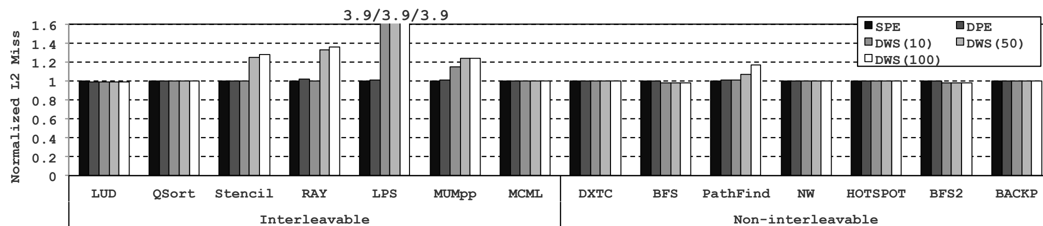
DWS, in general, can decrease idle cycles because it significantly increases Avg_{Path} to enable better interleaving. However, the significantly lower



(a) Number of idle cycles accumulated throughout all GPU cores.



(b) Number of L1 misses.



(c) Number of L2 misses.

Figure 3.10: Changes in idle cycles and L1/L2 cache misses when using different mechanisms (all normalized to SPE).

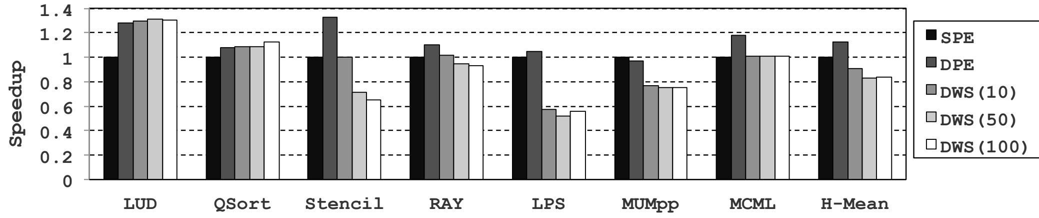
SIMD utilization achieved with DWS, makes a comparison of idle cycles between DWS and DPE meaningless. While the GPU executes instructions on more cycles, additional cycles are required to execute the many warp-splits of DWS. LUD and QSort can be directly compared because they have similar SIMD utilization with DWS and DPE, and also have similar total idle cycles.

Counter intuitively, RAY, LPS, PathFind, and HOTSPOT, which have significant improvements of TLP with DWS, suffer from many more idle cycles compared to SPE. The reason for this behavior is that the many interleaved warp-splits present a memory access pattern that performs poorly with the cache hierarchy. As shown in Figure 3.10(b–c), these four benchmarks have increased miss rates in both L1 and L2. The added TLP is not sufficient to counter-weigh the added memory latency. Stencil and MUMpp also suffer from worse caching with DWS, but have high-enough TLP to still reduce the number of idle cycles.

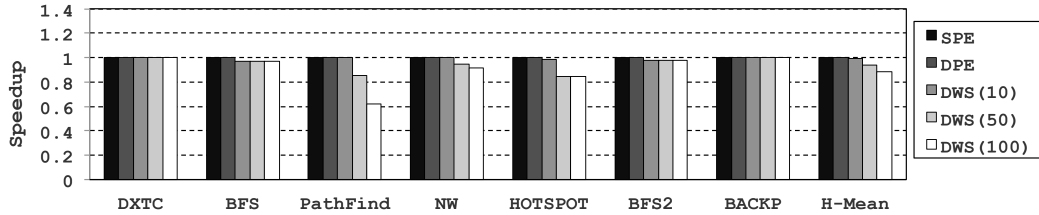
3.3.3.3 Overall Performance

Figure 3.11 shows the overall performance of DPE compared to that of SPE and DWS. Except for MUMpp, whose IPC is degraded by 3%, due to its increased L1 miss rate, DPE provides an improvement in performance across all the interleavable workloads (12.7% on average) while never degrading the performance of non-interleavable ones.

DWS is able to obtain significant IPC improvement for LUD and QSort (30.7%/12.2% increase over SPE and 2.4%/1.2% over DPE when using DWS_{50}), thanks to the significant increase in Avg_{Path} while maintaining similar SIMD lane utilization. The other 12 benchmarks, however, fail to balance Avg_{Path} and SIMD lane utilization and either suffer from degraded performance due to excessive subdivision or do not subdivide at all despite having potential for interleaving (MCML). It is interesting to observe that despite large increase in



(a) Speedup over SPE among interleavable benchmarks.



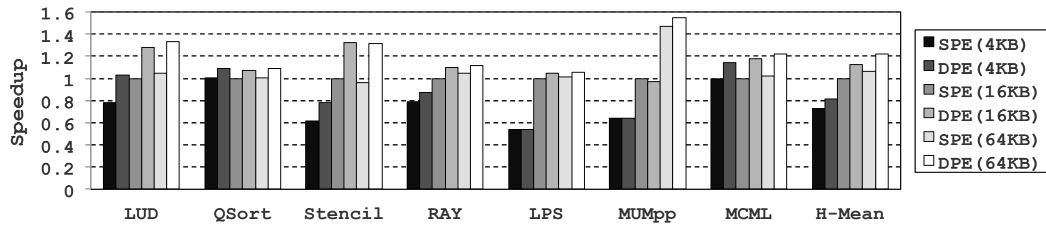
(b) Speedup over SPE among non-interleavable benchmarks.

Figure 3.11: Performance of the DPE model compared to SPE and DWS (all normalized to SPE).

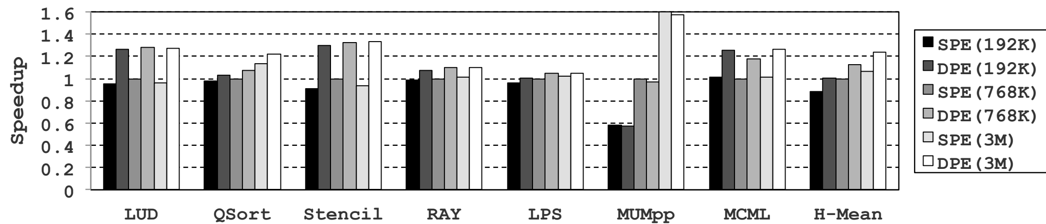
Avg_{Path} achieved with DWS, the significant loss in SIMD lane utilization always outweighed the benefits of increased interleaving. This is mainly because the increase in Avg_{Path} (and hence increased interleaving capability) is only beneficial upto the point where there exists any latency to hide, after which the loss in SIMD lane utilization is too severe. GPUs are designed to tolerate high latency, so this is, in fact, expected behavior.

3.3.3.4 Sensitivity Study

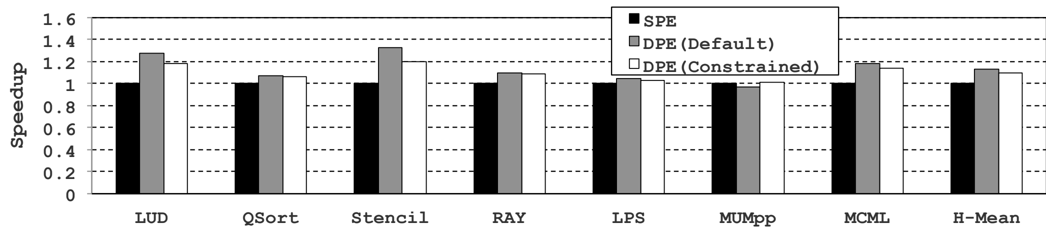
Figure 3.12(a–b) shows the speedup of DPE over SPE for the 7 interleavable benchmarks with different cache sizes. With smaller cache, we could expect a higher need for hiding latencies because of the larger fraction of long-



(a) Performance sensitivity to different L1 cache size (normalized to SPE(16KB)).



(b) Performance sensitivity to different L2 cache size (normalized to SPE(768K)).



(c) Performance sensitivity when warp scheduler has limited context resources (normalized to SPE).

Figure 3.12: Performance sensitivity to cache size and warp scheduler visibility.

latency memory operations. Overall, the relative IPC improvement remains stable within ±4% when varying the size of the L2 cache and ±2% for L1 cache size variation, with the exception of Stencil. When the L1 cache is reduced to 4KB, Stencil becomes much more memory bound, which results in a significant increase of idle time. In this case, while DPE can still reduces

idle cycles by the same absolute number of cycles, the relative improvement is smaller because the overall execution time is so large.

As discussed in Section 3.2, DPE with a more constrained scheduler and with a scoreboard that does not introduce false dependencies is also evaluated. The more aggressive scoreboard improved performance by at most 1%. *DPE (Constrained)* in Figure 3.12(c) can only track a single path's context per warp so the schedulable path is rotated (between $Path_L$ and $Path_R$ after a long-latency instruction executes). Overall, speedup is reduced from 12.7% to 9.9% for the constrained mode of the warp scheduler.

3.3.3.5 Implementation and Energy-Efficiency of DPE

As discussed in Section 3.2, implementing DPE requires modifications to the reconvergence stack, scoreboard, and the scheduler, as well as a minimal extension to propagate a single bit to indicate whether an instruction originated from the left or the right path. Each stack entry with DPE requires 160 bits to store the PC and mask of each path (32 bits each per path and the single RPC (32 bits)). While this is more bits per entry compared to the SPE stack, which requires 96 bits, fewer stack entries are needed to represent the same number of paths. The maximum stack depth observed with the 14 benchmarks we evaluated in detail was 11 with SPE and 7 with DPE, with a very similar overall size of structure. We therefore estimate that the DPE stack has negligible overheads compared to SPE.

The DPE scoreboard requires independent left and right scoreboards, the addition of the single shadow bit to each entry, and logic for setting the shadow bits and selecting whether the pending or shadow bit should be used when querying. The additional logic is very simple and should have minimal overhead. The extra shadow bit accounts for $7-14\%$ of the scoreboard storage, depending on the maximum number of registers per thread, which increased from 64 to 256 between NVIDIA's Fermi [38] and Kepler [30] GPUs. Maintaining the information for the two paths roughly doubles the cost of the scoreboard in area and power. While the scoreboard is significantly more expensive, such overhead is also a necessary component for DWS that seeks multi-path execution. The warp scheduler hardware also roughly doubles in size because decoded instructions from both left and right paths require instruction-buffer storage. Like the scoreboard, the scheduler is amortized across all lanes. Note that previous studies [52, 53] estimate the majority of intra-core power being consumed by the large register file (26% of intra-core power in GTX480, the baseline GPU architecture of this study), integer/floating-point execution units (38%), and SIMD pipelines (22%). The authors of the recently introduced GPUWattch [53], for instance, estimate the power overhead of the scoreboard to be negligible and omit its power due to its insignificance. These prior studies corroborate our estimation and we conclude that the energy-overhead of DPE to be negligible.

3.4 Discussion

This subsection summarizes some key discussion points that are related with the proposed design. We start by discussing four possible extensions to the DPE microarchitecture, fostering future research based on the intuition of DPE. We then discuss DPE's impact on programmability and conclude this subsection with a qualitative comparison against a recently proposed research project that is closely related with DPE.

3.4.1 Path-Forwarding

DPE exposes two paths for scheduling when the TOS entry has both a left and a right path. When one of these two paths reconverges and the other is still active, only a single path is available for scheduling. However, in some cases an independent path, which can be concurrently scheduled with the current active path, may exist in entries below the TOS. In the example shown in Figure 3.3(d), interleaving block B and block E does not break correctness, but is not done because block B is not at the TOS. A possible optimization of DPE to such issue is to *forward* the information from a lower stack entry up (including its RPC) when a slot at the TOS is available. The details of how this forwarding can be achieved with reasonable logic circuits is omitted, because the evaluation of such path-forwarding technique indicates a small overall potential for improvement; whenever the TOS entry contains only a single available, we exhaustively traverse down the stack and expose an interleavable path, if available, to the scheduler. We observed a maximum of $< 2\%$

performance improvement on the 7 interleavable benchmarks. The interleavable benchmarks tended to have a shallow stack and balanced branches, which limits the opportunities for forwarding. While proven ineffective in the studied benchmarks, path-forwarding will provide significant benefits when the taken and non-taken paths are not balanced with one path executing significantly longer than the other.

3.4.2 Multi-Path Execution Model

This study limits extending each stack entry to accomodate *two* paths simultaneously, because the bi-path (*if* and *else*) nature of branches smoothly fits with the stack model with minimum implementation overhead (Section 3.2). Having the stack track *multiple* paths at the TOS, however, can be done and may be interesting to explore. As mentioned in Section 3.4.1, there can be several cases where the DPE model still limits available TLP (e.g., one of the paths at the TOS is currently invalid but stack entries underneath the TOS still contain interleavable paths invisible to the warp scheduler). Extending the stack model to accomodate multiple paths at the TOS requires mechanisms to (a) have the warp scheduler constantly check multiple interleavable path information, and (b) have the scoreboard resolve multiple cross-path data dependency issues. Supporting the first issue requires searching/traversing through the stack entries underneath the TOS, which can be non-trivial when multiple stack entries exist (e.g., maximum stack depth observed was 11 with SPE and 7 with DPE). Resolving the second issue can also be non-trivial,

given the scoreboard changes that are required for DPE's dual-path data dependency tracking (Section 3.2.2). Note that the path-forwarding technique in Section 3.4.1 provided less than 2% benefit across the 7 interleavable benchmarks, alluding to very limited potential benefits of using more than two paths. Due to the limited potential and significant increase in complexity, we do not explore this mechanism further in this dissertation.

3.4.3 DPE for Memory Divergence

While DWS increases TLP when branches diverge, an important benefit it can provide is to increase memory-level parallelism when some threads in a warp experience a cache miss while others hit in the cache. When such a case occurs, the warp can be split into two groups of threads: those that completed the load and continue to execute and those that must wait for main memory to supply the value. DPE hardware can also be used to increase TLP in such cases by utilizing the left and right slots for the mask of those threads that completed the loads and those that did not. DPE is not as flexible as DWS because the stack must still correctly reflect control flow reconvergence, which means the memory-split warp must wait at the first divergent branch or when the RPC is reached. The evaluation and optimization of this use of DPE is left for future work.

3.4.4 DPE with a Software-Managed Reconvergence Stack

The baseline GPU architecture uses an explicit hardware reconvergence stack, which maintains the PC, mask, and RPC. One alternative architecture is to maintain only the PC and mask in hardware and control when entries are pushed and popped with explicit software instructions [42]. Applying DPE to this design is straightforward. The only change needed is that a pop instruction only actually pop the stack if the other path is not active; if both paths are active, the first pop instruction disables its path and the second pops the stack. Some current GPUs, such as the GPU of the Intel Sandy Bridge Processor [32], have an entirely implicit stack. Hardware maintains an explicit PC for each thread and dynamically computes predicate masks based on a software-managed warp-wide PC. To support DPE, two warp-wide PCs are required and the details of what the software algorithm required to do so might be is left for future work.

3.4.5 Impact on Programmability

For divergent control flow with both *if* and *else* clauses (Figure 3.7), the number of cycles where SPE unnecessarily throttles available TLP (among those that are ideally available) linearly increases as the number of instructions within both paths gets larger. Programmers that try to highly tune the kernel for maximal SIMD efficiency should hence be aware of SPE characteristics. This is because large *if*/*else* clauses lead to significant reduction in available TLP, which could limit scheduling opportunities. One key advantage

of DPE is that programmer does not need to worry about such a reduction in TLP, as the underlying microarchitecture will automatically expose the interleavable path information to the thread scheduler. Given that the current SIMT execution model requires programmers to deeply understand the GPU microarchitecture in order to maximize efficiency, support such as that of DPE allows the programmer to worry about one less thing.

3.4.6 Alternative to Stack-Based Reconvergence Model

In addition to DWS, the *dual-instruction multiple-thread* (DIMT) execution model has recently been presented [44, 54]. DIMT can issue two different instructions to the SIMD pipeline at the same time by expanding the instruction broadcast network, the register file structure, and others. Brunie et al. [44] explored the microarchitectural aspects of adopting the DIMT concept to GPU architectures. Their DIMT-based architecture is conceptually similar to DPE in that a maximum of two concurrent paths are chosen for scheduling. Also like DPE, the scoreboard and scheduler are enhanced to track the larger number of schedulable units. Unlike DPE, DIMT does not work with the stack model. Instead, the more complex model of *thread frontiers* [55] serves as the baseline architecture. Support for thread frontiers requires significant changes to the hardware, including explicit tracking of per-thread PCs, new instructions, compiler support, and a hardware-managed heap structure that takes the place of the simple reconvergence stack. In addition, DIMT introduces a complex scoreboard design with significant additional storage, and new logic

functionality. While thread frontiers have advantages over the stack model for applications that make heavy use of unstructured control flow, they do present a more complex design point. Note that thread frontiers do not change the execution of structured control flow, which DPE primarily focuses on. DPE, in contrast, integrates smoothly with current execution models and designs and extends the reconvergence stack rather than replacing it. DPE is the first microarchitecture that is able to utilize intra-warp parallelism of this type with a reconvergence stack. DWS uses the stack only until warps are split and then abandons the design until a warp is merged again, and DIMT assumes the heap-based threads frontier model. In fact, Brunie et al. [44] explicitly state that a motivation for adopting thread frontiers in their design is that intra-warp TLP is very challenging with the stack model. This dissertation shows that intra-warp parallelism can still be achieved while maintaining the simple, elegant stack model without substantial modifications to the GPU processor architecture.

3.5 Summary

In this chapter we explored the potential for utilizing the intra-warp parallelism resulting from diverging structured control flow to improve SIMD efficiency and overall performance. DPE is the first mechanism that maintains the elegant control flow execution of the GPU reconvergence stack, yet is able to exploit intra-warp parallelism. Unlike prior approaches to this issue, DPE does not require an extensive redesign of the microarchitectural components,

and instead extends the stack to support two concurrent execution paths. The scoreboard and scheduler must also be enhanced, and we show how this can be done relying mostly on replicating current structures rather than adopting a completely new model. This chapter demonstrates that the combination of these spot-enhancements can provide significant efficiency and performance benefits and never degrades performance compared to the baseline GPU architecture. The maximum speedup across the studied benchmarks is 32% with an average of 12.7%. The potential improvements to our design with more aggressive and less constrained hardware are also discussed. My dissertation does not evaluate these in detail because the potential high cost and complexity of these modifications yields little performance improvement. The reason is that, with DPE, the additional latency hiding capability is already significant, and additional minor increases are insignificant, improving performance by no more than an additional 2%.

Chapter 4

Enhancing Compute Resource Utilization

So far, we have discussed restrictions on available thread-level parallelism due to SIMT control divergence and how DPE is able to alleviate such inefficiency. Another significant challenge with control divergence is the underutilization of SIMD units. This chapter first reviews a previously proposed research project, *thread-block compaction*[1], which seeks to improve SIMT compute resource utilization upon irregular control flow. The advantages as well as the limitations of this technique are thoroughly analyzed and are followed by a description of two novel optimizations to thread-block compaction, namely *compaction-adequacy prediction* [25] and *SIMD lane permutation* [26].

4.1 Thread-Block Compaction for Irregularity

As discussed in previous chapters, the SIMT model enables each thread to maintain its own logical control flow. The hardware generated active bitmasks (predicates), designate whether a thread is active or not, allowing independent branching for each thread. Because threads that are masked out do

[1] In this chapter, we use the term *thread-block* and *concurrent thread array* (CTA) interchangeably.

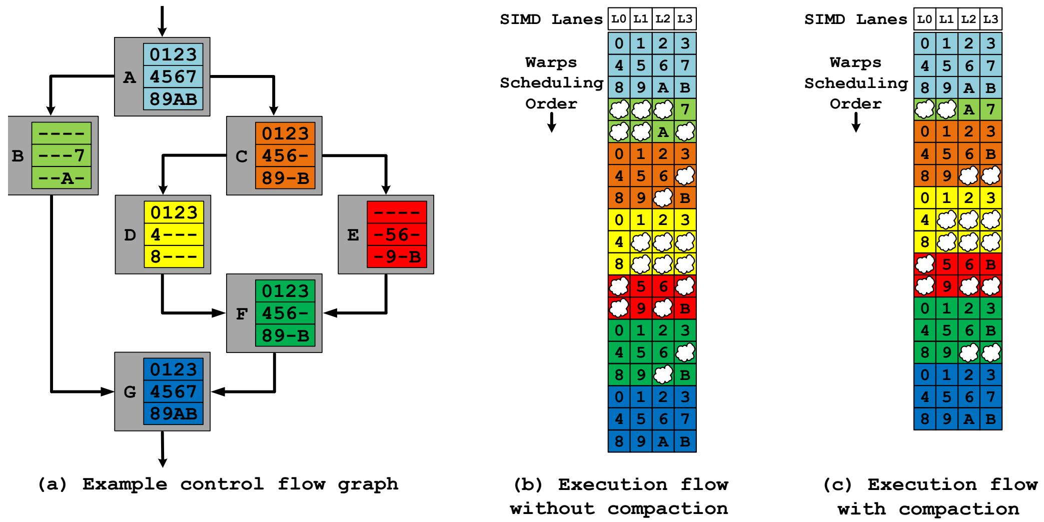


Figure 4.1: Example showing how a control flow graph (a) is executed without compaction (b) and with compaction (c). Each CTA contains 3 warps of 4 threads each. The numbers represent the thread-IDs that are executing in a basic block, while "–" denotes masked-out threads.

not commit the results of their computation, only threads that are active actually execute the instruction; thus enabling SIMT, however, partially serializes execution on a divergent branch as the true and false path must be executed one after another with those threads on the non-active path being masked out (shown as "bubbles" in some SIMD lanes in Figure 4.1). Accordingly, each divergence reduces SIMD lane utilization because more threads are masked out from execution.

4.1.1 Irregular Control Flow and SIMD Underutilization

The stack-based reconvergence model, as described in Chapter 2, can alleviate the resource underutilization by having the threads from the true

and false path reconverge at the immediate post-dominator [22, 41] (the first instruction of basic block F and G in Figure 4.1(a)) in the control flow graph. Figure 4.1(b) provides a high-level overview of how warps are scheduled when executing the control flow graph of Figure 4.1(a) without thread-block compaction (No_TBC), which is identical to the baseline SPE model[2] as discussed in Chapter 3. While the divergent branches at the end of block A and C reduce SIMD lane occupancy, the inefficiency is minimized by reconverging the diverged paths at the immediate post-dominator (block F and G).

4.1.2 Thread-Block Compaction

Although the baseline stack-based reconvergence model correctly handles even nested divergent branches, each divergence reduces the number of active threads and increases the number of wasted execution slots. Several mechanisms have been proposed to mitigate the magnitude of waste, including those that seek to *dynamically* combine active threads from multiple warps and schedule them with better SIMD efficiency [22, 23, 24]. Among these, compaction-based architectures [23, 24] have been actively studied and we detail the intuition behind compaction below.

The basic idea of improving utilization through compaction is to consider multiple warps, up to the entire thread-block (or *concurrent thread-arracy* (CTA)), as a single unit when a branch diverges and reconverges. Instead of

[2]In this chapter, we refer to the baseline SPE model as No_TBC to clearly differentiate it from TBC.

allowing each warp to be serially executed, threads executing in a common basic block dynamically form new warps to minimize masked execution slots. *Thread-block compaction* (TBC) [23][3], for instance, considers the entire CTA as a single unit when a branch diverges. TBC considers the active mask of the entire CTA and dynamically compacts it at all diverging paths (true/false paths) and reconvergence points. To maintain hardware efficiency, compaction is activated while maintaining the *fixed* association of each thread to its home SIMD lane. The two threads executing in Figure 4.1–path B (thread-ID = 7 and A), for instance, can be compacted into a single warp; as both threads are executing in different SIMD lanes, SIMD lane utilization is improved. The active threads executing in path C, however, are not compactable as thread-ID 0, 4, and 8 are all aligned in the leftmost lane, preventing compaction. Compaction is performed by a set of priority encoders that leverage the *CTA-wide* active bitmask to identify the minimum number of warps to execute the active threads. The microarchitectural components of TBC is detailed in Section 4.1.3. When the number of active warps generated through compaction (NW_{TBC}) is smaller than the number of warps needed to execute without compaction (NW_{NoTBC}), SIMD lane utilization is improved (e.g., path B in Figure 4.1(c)).

[3]Fung and Aamodt proposed TBC as an optimization to their previously suggested technique, *dynamic warp formation* (DWF) [22]. TBC resolves several limitations of DWF, so the proposed ideas in my dissertation are directly compared against TBC.

4.1.3 TBC Microarchitecture

The TBC mechanism is a low cost and elegant mechanism to dynamically reform warps. The basic idea of improving utilization in TBC is to consider the entire CTA as a single unit when a branch diverges. TBC considers the active mask of the entire CTA and compacts it, when possible, into a smaller number of warps with fewer masked threads. This is done by changing the reconvergence stack architecture from one stack per warp to a single stack for the entire CTA (Figure 4.2). This single stack is used to determine the point when the active masks of the true and false path are fully known in order to apply maximal compaction. TBC, in effect, uses its stack structure to introduce a *barrier* after each branch so that the compaction unit has access to the *entire* active mask of the CTA. This hardware-induced barrier is meant to enable performance gains but is not needed for correctness. A similar approach was suggested by Narasiman et al. [24], except that the granularity for compaction is a fixed *long warp*, which is decoupled from the CTA size, and that unconditional branches do not stall and do not wait for the entire CTA to reach the jump point. In Section 4.4, my proposed ideas are compared with TBC, as well as an optimized version of TBC (TBC+) that does not stall on either unconditional branches (as in [24]) or on conditional branches that are guaranteed not to diverge. Such non-divergent branches are identified by the CUDA compiler (marked with a `.uni` qualifier in PTX [56]).

Although a single active mask is maintained per CTA, warps are still scheduled independently by the warp scheduler based on available resources

- CTA_n : CTA with CTA-ID *n*
- BR_{x-y} : PC value of a branch instruction that diverges into path *x* and *y*

(CTA_0)

	RPC	Active Mask	PC	WCnt
TOS→	-	0123 4567 89AB	A	3

(a) Initial stack status of the CTA-wide reconvergence stack. All three warps are active with a full bitmask.

(CTA_0)

	RPC	Active Mask	PC	WCnt
TOS→	-	0123 4567 89AB	A	2
	G	0123	C	0
	G	----	B	0

(b) Branching path information of W_0 is updated to the corresponding active mask field after the branch instruction at A_2 is executed.

(CTA_0)

	RPC	Active Mask	PC	WCnt
TOS→	-	0123 4567 89AB	A	1
	G	0123 456-	C	0
	G	---- ---7	B	0

(c) Likewise, when W_1 arrive at the end of basic block A (BR_{B-C}), the corresponding path information is updated to the active mask field.

(CTA_0)

	RPC	Active Mask	PC	WCnt
TOS→	-	0123 4567 89AB	A	0
	G	0123 456- 89-B	C	0
	G	---- ---7 --A-	B	0

(d) After W_2 arrives at BR_{B-C}, no warps are available to execute at path A, so execution transitions to basic block B.

(CTA_0)

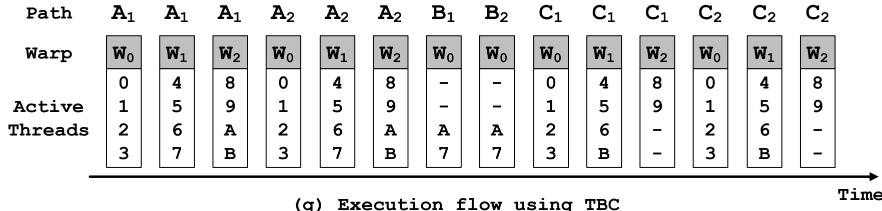
	RPC	Active Mask	PC	WCnt
	-	0123 4567 89AB	A	0
	G	0123 456- 89-B	C	0
TOS→	G	---- ---7 --A-	B	1

(e) Compaction unit uses CTA-wide bitmask at path B to combine thread-ID 7 and A into a *single* warp. TOS now points to path B.

(CTA_0)

	RPC	Active Mask	PC	WCnt
	-	0123 4567 89AB	A	0
TOS→	G	0123 456- 89-B	C	3

(f) Stack entry at TOS is *popped out* when the compacted warp at basic block B arrives at the end of path B.



(g) Execution flow using TBC

- Path X_Y : *Y*-th instruction in basic block X
- Warp W_n : Warp with Warp-ID *n*

Figure 4.2: TBC microarchitecture and its high-level operation. Figure assumes the same control flow graph in Figure 4.1, except that each basic block consists of two instructions.

and operands. Unlike SPE, however, the single entry for the entire CTA implies that all active threads across all warps are all executing instructions

from within a single basic block. All active (unmasked) threads execute the same sequence of instructions until they reach another branch, at which point they must wait for all warps to reach the branch to allow another compaction, or until they reach a reconvergence point. To achieve this, hardware needs to know when all active warps have reached a branch or are ready to reconverge. Both branching and reconvergence can only occur after all active warps have synchronized.

Accordingly, in addition to the PC, active mask, and reconvergence PC fields, each entry of TBC's stack also includes a counter for determining the number of active warps in the current control flow path (WCnt). When the TOS is changed, the new TOS entry's WCnt is initialized to the number of compacted warps associated with the entry (Figure 4.2(a,e,f)), which is equivalent to the minimum number of warps that are executing along that basic block. Each time a warp executes a branch or reaches a reconvergence point, WCnt of the TOS is decremented (Figure 4.2(b–d)). When WCnt reaches zero, all active warps have reached the end of the basic block and executed the branch instruction or reached the reconvergence point. The first time a branch diverges, new entries are added to the stack for the false and true paths (Figure 4.2(b)). Note that the TOS is not changed because the entire CTA advances through the control flow graph in unison. The true and false active masks for the warp that just executed the branch are then added to the new entries. Each additional warp that executes the branch incrementally updates its masks as well. When WCnt becomes zero, the next basic block can

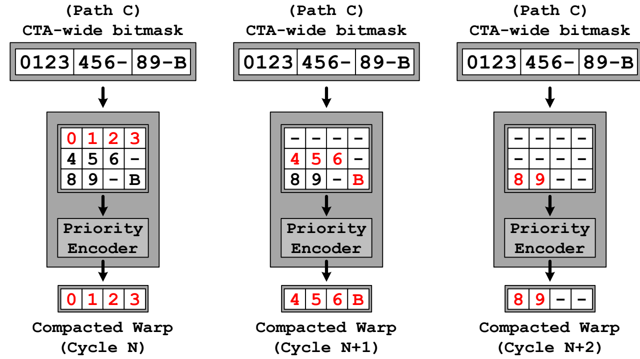


Figure 4.3: Example showing how the priority encoders compact the CTA-wide bitmask (at basic block C of Figure 4.1) into three warps ($NW_{TBC} = 3$). Although we explicitly use the thread-IDs to illustrate the bitmask, the actual hardware uses single bits. Note that threads with thread-ID 0, 4, and 8 always execute in the leftmost SIMD lane, as each thread's home SIMD lane is statically determined using their thread-IDs. Compaction is ineffective in the above example as NW_{TBC} equals NW_{NoTBC}.

execute, at which point the TOS is updated to point to the true path of the branch and the CTA-wide active mask is ready for compaction (Figure 4.2(e)).

Compaction is performed by the *warp compaction unit* (WCU), which is shown in Figure 4.3. The WCU receives the CTA-wide active mask, when WCnt becomes zero. The WCU then uses a set of priority encoders to identify the minimum number of warps required to execute the active mask while maintaining the *fixed* association of each thread to its SIMD lane. This is necessary to avoid high-overhead changes to the register file [57, 22, 23, 24], and allows active threads to be compacted into a common warp as long as they are executing in different SIMD lanes. The output of the WCU is a new set of active masks and the number of warps needed to execute them. This

information is then used for the new TOS entry to continue execution. The context information of how the original threads are associated with the newly compacted warps, the thread-ID mapping, and the individual warp PC values are stored and maintained by the warp scheduler as detailed by Fung and Aamodt [23].

4.1.4 Limitation of TBC

When the WCU successfully compacts the CTA-wide active mask into fewer warps than would have executed with baseline SPE, performance is likely to improve as illustrated in Figure 4.1. Although TBC introduces more synchronization points and forces all warps within a CTA to execute in the same basic block, when there are warps available for scheduling from its own CTA or from other CTAs, the synchronization overhead can be effectively hidden. As we show in Section 4.4.2, however, in many cases compaction does not result in reducing the number of warps and parallelism is often limited. In other words, previous compaction mechanisms are often an overkill as control divergence most commonly occur in a *non-compactable* manner, especially for workloads that rarely experience divergence. Yet, because they provide no means to differentiate compaction-ineffective branches, the hardware logic units associated with compaction (such as the WCU) are always activated for all branching points – which in turn consume power unnecessarily for compaction-ineffective branches. In the following section, the proposed *compaction-adequacy predic-*

```
Example) Code that contains two potentially divergent branches.

0    /*    - Excerpted from __global__ void bpnn_layerforwarrd_CUDA( ) kernel of BACKP.
1          - float input_cuda[ ] designates a global memory region */
2
3    int   by    = blockIdx.y;
4    int   tx    = threadIdx.x;
5    int   ty    = threadIdx.y;
6
7    __shared__ float  input_node[HEIGHT];
8
9    if ( tx == 0 )       // Conditional branch that is divergent but compaction-ineffective
10      input_node[ty] = input_cuda[index_in];
11
12   __syncthreads();
13
14   …
15   for ( i = 1; i < __log2f(HEIGHT) ; i++ ){
16         …
17                        // Loop-end branch: Conditional branch that is non-divergent
18   }
```

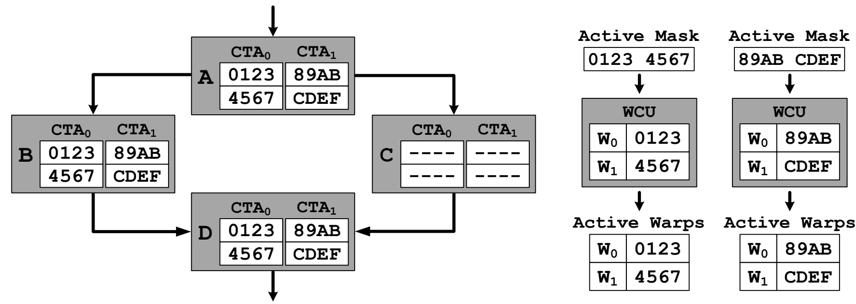
Figure 4.4: Example source code containing conditional branches that are either non-divergent or divergent but compaction-ineffective. *BACKP* is part of the Rodinia benchmark suite [14].

tor microarchitecture is described that alleviates this unnecessary synchronization problem.

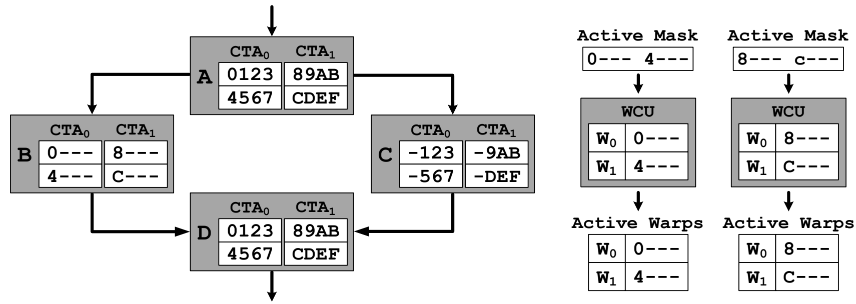
4.2 Compaction-Adequacy Prediction

4.2.1 Motivation and Key Insights

This work is motivated by the key observation that not all branch instructions are likely to benefit from compaction because conditional branches often follow patterns that are repeated between warps. For example, many applications operate on a large loop that processes target arrays, but depending on the number of threads/warps accessing the array the elements at the edges/corners are treated differently. Some of these are handled by adding *if*-statements inside the main loop body, which are conditional but typically



(a) The loop-end branch on line 17 (non-divergent) of Figure 4.4.



(b) The divergent yet compaction-ineffective branch on line 9 of Figure 4.4.

Figure 4.5: Control flow graphs of compaction-ineffective branches.

non-compactable. Also, a common software optimization is to construct kernels such that all threads in any given warp branch in the same way, which effectively eliminates branch divergence, even though the conditional branch is still dynamically determined. A simple example is a loop-end branch, which is conditional but is most typically evaluated to be the same across all threads (line 17 of Figure 4.4 and Figure 4.5(a)). In other cases, conditionals are not well optimized or some threads are intentionally masked to reduce memory traffic. In such cases, a divergent branch may diverge in a similar way across all warps, which renders compaction ineffective. An example of a con-

ditional branch that diverges but is compaction-ineffective is shown in line 9 of Figure 4.4 and in Figure 4.5(b). Several of the benchmarks we evaluate (BITONIC, REDUCT, LPS, BACKP, FDTD3D, 3DFD, and QSRDM) encounter a significant number of branches that cause divergence in a regular pattern across warps and as a result cannot be compacted; the number of warps before and after compaction is the same.

As we discuss in Section 4.4, attempting an ineffective compaction can potentially reduce performance, especially when a kernel contains a significant number branches that are mostly compaction-ineffective. In order to collect candidates for compaction, previous mechanisms stall all threads in the CTA until the last thread reaches the branch point. At that time, compaction occurs and the newly compacted warps can be scheduled. This compaction-induced barrier introduces synchronization overhead, which cannot always be hidden. While the benefits of compaction usually outweigh the overhead of synchronization when successful, that is not the case when compaction is ineffective. When unsuccessful, compaction merely result in shuffling the threads between warps which can potentially cause memory divergence (Figure 4.6) with no benefits in SIMD utilization and worsening power efficiency through needlessly activated compaction units. The goal of CAPRI is to overcome this inefficiency by stalling only those warps that have a high likelihood of benefiting from compaction. Warps that are not likely to gain from compaction are *bypassed* so that they can continue execution beyond the branch while maintaining their static warp structure.

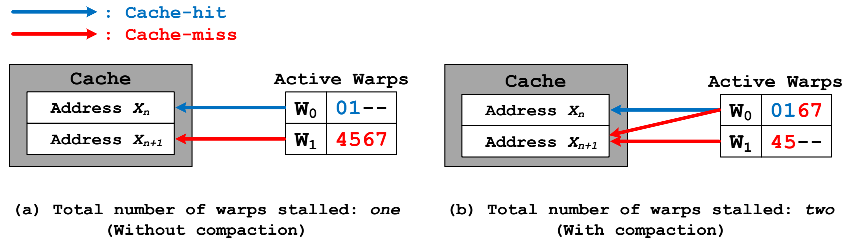


Figure 4.6: Example of a non-compactable warp that leads to memory divergence after compaction.

This goal is achieved by adopting a *compaction-adequacy predictor* (CAPRI) [25], which uses the active mask of a warp, just after a branch point, and its adequacy history to predict whether a warp should wait for compaction or continue to execute, ignoring potential compaction opportunities. The adequacy history is the history of successes or failures of compaction with respect to a particular branch. To predict compaction-adequacy, CAPRI follows a design similar to a simple single-level branch predictor [58]. CAPRI uses a prediction table that tracks adequacy history using prediction bits, which are updated based on a dynamically computed compaction-adequacy of conditional branches. Compaction-adequacy is computed, regardless of whether compaction was applied to the warps or not, because warps that did not stall for compaction could have benefited from compaction.

CAPRI consists of three main components: the *compaction-adequacy prediction table* (CAPT) that tracks adequacy history, the *decision logic* that determines whether to delay a warp for compaction, and the *WCU* which includes a logic unit that determine the compaction-effectiveness of a branch.

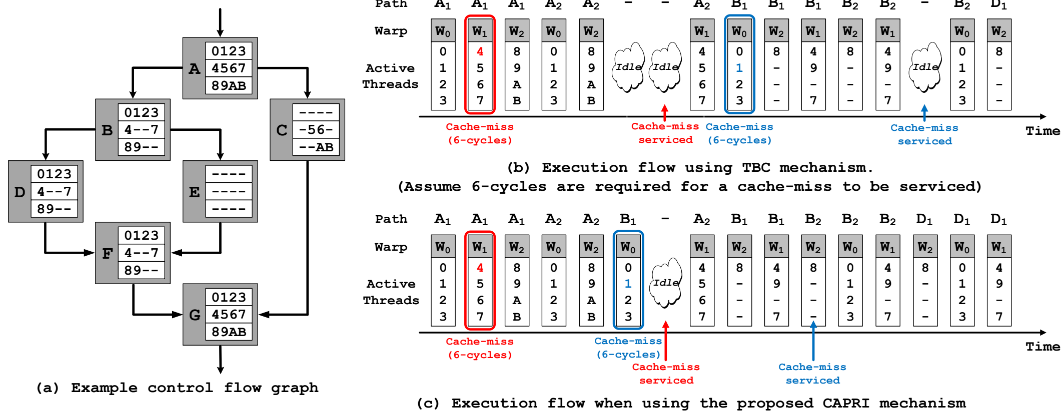


Figure 4.7: (a) Example control flow graph and its corresponding execution flow when (b) TBC or (c) CAPRI is used. Each basic block consists of two instructions and each CTA consists of 12 threads, 4 threads per warp.

Figure 4.7 is used as an example to illustrate how TBC and CAPRI execute the control flow graph in Figure 4.7(a), which contains a non-divergent branch and a branch that is divergent but compaction-ineffective. Notice that with TBC, each branch instruction introduces a barrier, which in the example leads to three idle cycles because the barrier restricts the parallelism available to hide memory latency (Figure 4.7(b)). With CAPRI, on the other hand, warps that encounter branches that are compaction-inadequate do not wait and no barrier is introduced, effectively reducing the number of idle cycles to one. In general, CAPRI enables greater scheduling flexibility and better latency hiding, which improves performance (Figure 4.7(c)). We will refer back to this example when explaining CAPRI's components and operation below.

4.2.2 CAPRI Microarchitecture

Compaction-Adequacy Prediction Table. The CAPT is implemented with a fully-associative (potentially set-associative) tagged structure used to track adequacy history for branches (Figure 4.8). Each CAPT entry consists of a tag that identifies a particular branch using the PC of the branch instruction (BADDR), one or more adequacy history bits, and a valid bit. As we show later, the maximum number of CAPT entries necessary in all the evaluated benchmarks is 24, and in practice, a 8-entry CAPT was capable of achieving 97% of the benefits provided by an infinite CAPT among the benchmarks that were sensitive to the number of entries. Note that because of its small size, we maintain a separate CAPT for each GPU core. We experiment with several configurations of the history bits of each CAPT entry: a 2-bit saturating counter, a single bit indicating the last seen compaction-adequacy behavior, and a single sticky bit that is set once a warp diverged upon a branch instruction.

Decision Logic. There are three possible outcomes when a warp executes a branch. The first is that the warp did not diverge and there is no benefit to stalling. Therefore CAPRI's policy is to skip checking the CAPT and simply allow a non-divergent warp to continue executing. This scenario is shown in Figure 4.8(a). Even without prediction, the fact that non-divergent warps are not stalled already provides an advantage over previously proposed compaction schemes, but is not the main contribution of CAPRI.

TOS	RPC	Active Mask	PC	WCnt
TOS→	-	0123 4567 89AB	A	2
	G	----	C	0
	G	0123	B	1

Warp Scheduler: Ready Warps List			
Status	Ready	Stalled	Ready
TID	0123	4567	89AB

WCU	
W_0	----
W_1	----
W_2	----

Valid	History	BADDR
0	0	-
0	0	-
0	0	-

(a) Because W_0 does not diverge upon arriving at PC=BR_{B-C}, W_0 is *bypassed* and increments WCnt of path B by one. UMask is updated to (011) to have W_0's active mask not be considered for generating compacted warps.

TOS	RPC	Active Mask	PC	WCnt
TOS→	-	0123 4567 89AB	A	1
	G	---- --AB	C	0
	G	0123 89--	B	1

Warp Scheduler: Ready Warps List			
Status	Ready	Stalled	Invalid
TID	0123	4567	-

WCU	
W_0	----
W_1	----
W_2	----

Valid	History	BADDR
1	1	BR_{B-C}
0	0	-
0	0	-

(b) W_2 arrives at PC=BR_{B-C} and is *stalled* upon divergence. WCnt of path A is decremented by one and UMask remains at (011) because the active mask for W_2 needs to be considered for compaction.

TOS	RPC	Active Mask	PC	WCnt
	-	0123 4567 89AB	A	0
	G	---- -56- --AB	C	0
TOS→	G	0123 4--7 89--	B	1

Warp Scheduler: Ready Warps List			
Status	Ready	Invalid	Invalid
TID	0123	-	-

WCU	
W_0	0123
W_1	4--7
W_2	89--

Valid	History	BADDR
1	1	BR_{B-C}
0	0	-
0	0	-

(c) WCnt becomes zero as W_1 arrives at PC=BR_{B-C}. The CTA-wide active masks at TOS is forwarded to WCU and compaction is initiated. Note that W_0's active mask is not considered for compaction as UMask is (011).

TOS	RPC	Active Mask	PC	WCnt
	-	0123 4567 89AB	G	0
	G	---- -56- --AB	C	0
TOS→	G	0123 4--7 89--	B	3

Warp Scheduler: Ready Warps List			
Status	Ready	Ready	Ready
TID	0123	49-7	8---

WCU	
W_0	----
W_1	----
W_2	----

Valid	History	BADDR
1	0	BR_{B-C}
0	0	-
0	0	-

(d) WCU generates two warps and increments WCnt by two. As the *Predictor* evaluates PC=BR_{B-C} to be compaction-ineffective, the history bit is reset. UMask is initialized back to (111).

Ready : Warp is able to be scheduled.
Stalled : Warp is waiting for a long-latency operation.
Invalid : Warp arrived at a branch and is stalled for compaction.

Figure 4.8: Example CAPRI execution (1b-Latest) of the control flow graph of Figure 4.7.

The second possible outcome is that the warp diverged and that the CAPT has no information about the divergent branch (Figure 4.8(b)). In this case, the branch is conservatively assumed to be an adequate candidate for compaction. The CAPT is updated to include this branch and the history is initialized to an *"adequate"* state. At this point, the CAPT has infor-

mation about the branch, which indicates that future warps should wait for compaction as well.

The third possible outcome is that the warp diverged and the corresponding branch is already registered in the CAPT. In such a case, the history bits of the CAPT are used to decide whether to stall the warp or to bypass compaction (Figure 4.8(c)). Accordingly, all warps that diverge from this CTA will follow the same CAPRI decision because the CAPT is only updated once the WCU evaluates compaction effectiveness and because history is maintained per branch as discussed below. Note that, as a design optimization, warps that have bypassed compaction are not permitted to execute more than a single basic block away from one another.

WCU, CAPT, and Reconvergence Stack Management. Regardless of whether the warps have been stalled or not, in order to evaluate the compaction effectiveness of a branch, the CTA-wide active mask at the TOS is always forwarded to the WCU when WCnt is zero. This is because warps that were predicted as compaction inadequate and did not wait for compaction could have, in fact, benefited from compaction. We follow the WCU design described by Fung and Aamodt [23] and only make one minor change: warps that did not wait for compaction are marked as *bypassed* using a single bit per warp (collectively referred to as the update-mask (UMask) in Figure 4.8). The WCU does not consider the active masks of the bypassed warps when compacting the warps, but the logic unit that derives the compaction effectiveness (*Predictor*

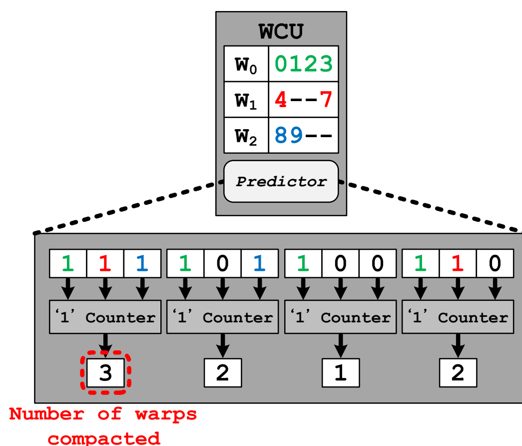


Figure 4.9: Evaluating the compaction-adequacy of the branch entering basic block B from Figure 4.7(a).

in Figure 4.9) still processes the corresponding active masks. An example of this is shown in Figure 4.8(c) to (d) and Figure 4.9.

The predictor, which evaluates compaction-adequacy inside the WCU, is simpler than the logic used to form compacted warps – it just counts the number of warps that the WCU would have output had all warps waited for compaction. Figure 4.9 shows this predictor logic that counts the number of active threads associated with each SIMD lane. The maximum number of threads is equal to the minimum number of required warps. If this minimum is equal to the number of active warps, no benefit is provided even though the warps were stalled. In such a case, this branch would not be adequately compacted and we update the CAPT to reflect that. If, on the other hand, the number of post-compaction warps is smaller, the CAPT is updated to indicate adequacy so that the warps would wait for compaction in future iterations.

I experiment with multiple history bit configurations: 2-bit saturating counter, 1-bit latest adequacy result, and 1-bit sticky adequacy. Figure 4.8 shows how the 1-bit latest history configuration (1b-Latest, 1bL) is updated to reflect the most recent adequacy result from the WCU. The 2-bit counters work in a similar manner, incrementing and decrementing the counter when the WCU evaluates a branch to have been effectively and ineffectively compacted, respectively. The 1-bit sticky configuration (1b-Sticky, 1bS) is the most conservative scheme. It sets the history bit to adequate when the warp that executed a branch diverges and the bit remains set until the kernel completes.

4.2.3 Summary of the Benefits of CAPRI

Previously proposed compaction mechanisms [22, 23, 24] fall short because they introduce excessive synchronization, even when compaction provides no benefits. Section 4.4.2 quantitatively demonstrates that a large fraction of conditional branches cannot benefit from compaction because they happen not to diverge much or because their divergence pattern is not amenable to compaction. CAPRI helps to overcome this fundamental deficiency of compaction and provides superior performance improvements when improvements are possible. At the same time, by correctly identifying the lack of compaction-adequacy, CAPRI matches the performance of the baseline No_TBC to within ±2%, substantially improving the robustness of compaction across a wide range of applications. As detailed later in this chapter, the implementation

overhead is trivial with each CAPT consuming less than 10mW (less than 0.3% of the 2.7W consumed by each GPU core in the Quadro FX5800 [53])[4] in all the applications that have been studied.

4.3 SIMD Lane Permutation

Section 4.2 described the proposed CAPRI microarchitecture that can significantly remedy the deficiencies of TBC. While CAPRI improves the robustness of compaction, I observe that there are still unexploited opportunities for compaction which CAPRI is not able to address. We start by discussing the limitations of CAPRI-enabled compaction and then describe the proposed *SIMD lane permutation* mechanism.

4.3.1 Current Compaction Limitations

While CAPRI is useful for alleviating the synchronization overhead of compaction, thus avoiding performance loss, it does not improve SIMD resource utilization compared to TBC. Figure 4.10 shows the average SIMD lane utilization ($SIMD_{util}$) of several benchmarks in Table 4.2, which exhibit branch divergence, using each of three configurations: baseline without compaction (*No_TBC*), TBC, and ideal compaction (TBC_{ideal}). With ideal compaction, threads can change SIMD lanes for optimal compaction. Changing SIMD lanes is not practical because of the required interconnect in the register

[4] Leng et al. [53] estimate that 48.1% of the Quadro FX5800's 171W power is consumed by the 30 GPU cores. We therefore assume each GPU core consumes 2.7W.

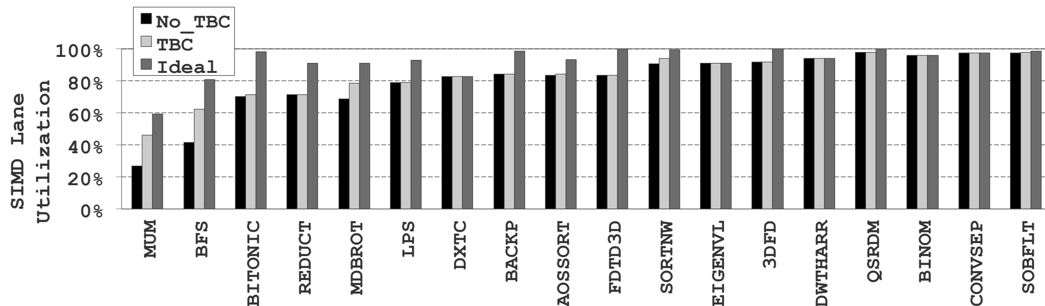


Figure 4.10: Average SIMD lane utilization of divergent benchmarks (18 among 20 that are listed in Table 4.2), without compaction (No_TBC), with a current branch compaction technique (TBC), and with ideal compaction (TBC_{ideal}). SIMD lane utilization is defined as the fraction of SIMD lanes occupied (active) when a warp is executing. In order to isolate the effect of idle cycles, we only average the SIMD lane utilization of issued warps with at least a single thread active.

file [57, 22, 23, 24], but TBC_{ideal} provides an upper bound on any compaction within a CTA.

Overall, 11 of the 18 benchmarks show noticeable improvement in $SIMD_{util}$ with ideal compaction (another 3 have improvement of less than 2%). Interestingly, while all applications with low $SIMD_{util}$ can benefit from compaction, even some applications with relatively high $SIMD_{util}$ show significant potential (e.g., LPS, BACKP, AOSSORT, FDTD3D, SORTNW, and 3DFD). It is also important to note that TBC is unable to approach the full potential of compaction for any of the benchmarks. Furthermore, some of the high-$SIMD_{util}$ benchmarks actually benefit more from the current TBC implementation than some with low $SIMD_{util}$. FDTD3D, for instance, shows a 17% benefit compared to just 13% exhibited by LPS.

While a benchmark's absolute $SIMD_{util}$ is certainly correlated with its compactability, it is not the sole factor that determines it. Rather, *how* the divergence manifests among warps is more critical because only threads that have diverged to the same path *and* do not share a common home SIMD lane can be compacted together (Figure 4.3). Previous compaction-mechanisms [23, 24, 25] are therefore only effective on divergent branches that do not cause active threads to align with a few common SIMD lanes (which we refer to as *aligned divergence* in the rest of this thesis). For example, the three active threads in Figure 4.3 are fully aligned at the leftmost SIMD lane. As a result, compaction provides no benefit and NW_{TBC} and NW_{NoTBC} both equal 3. Such aligned branches are fairly common, as we demonstrate in the rest of this chapter.

4.3.2 Aligned Divergence

What fundamentally decides the control flow of each thread is how the branch predicate is evaluated. *Aligned divergence* is a phenomenon where threads with a common home SIMD lane have their predicate condition resolved the same way, causing active threads to be concentrated on a subset of the SIMD lanes. I observe that such alignment is rarely exhibited when the predicate depends on input data arrays. In such data-dependent branches (*D-branches*), different threads most likely reference different values thereby resolving the predicate differently (Figure 4.11(a)). The divergence behavior of branches with a predicate condition that does not depend on a data array

```
Code #1) Branch depending on data arrays - (i)

0    // Code snippet from the kernel of BFS benchmark
1    // g_graph_visited and g_graph_edges are data array parameters.
2
3    int tid = blockIdx.x*MAX_THREADS_PER_BLOCK + threadIdx.x;
4
5    …
6    int id = g_graph_edges[…];
7    if( !g_graph_visited[id] )
8    {
9         …
10   }
```

(a) Threads that load a data array value (*g_graph_visited*) of zero execute the true path. The branch at line 7 is therefore data-dependent.

```
Code #2) Branch depending on data arrays - (ii)

0    // Code snippet from the kernel of SORTNW benchmark
1    // s_key[ ] and s_val[ ] are data array parameters.
2
3    uint ddd = dir & ((threadIdx.x&(size/2)) != 0);
4    …
5    Comparator(s_key[~], s_val[~], s_key[~], s_val[~], ddd);
6    …
7    __device__ inline void Comparator(uint& keyA, keyB, uint dir ){
8       if( (keyA>keyB) == dir ){ … };
9    }
```

(b) The intermediate variable (*ddd*), which is tainted by a programmatic value, is combined with values from data arrays (*s_key, s_val*) to calculate the predicate. As the data values referenced by each thread are likely different, the branch at line 8 is data-dependent.

Figure 4.11: Example kernel codes containing D-branches.

value (this thesis refer to such non-data array values as *programmatic* values) is substantially different from the behavior of D-branches.

A *programmatic* value is viewed the same way across the threads. The indices for the CTA-ID (`blockIdx`), width and height of a CTA (`blockDim`), and scalar input parameters of a CUDA kernel (e.g., *imageW* in Figure 4.12(b)),

Code #3) Branch dependent on a *programmatic* value - (i)

```
0    // Code snippet from the kernel of BITONIC benchmark
1
2    const unsigned int tid = threadIdx.x;
3    …
4    for (unsigned int k = 2; k <= NUM; k *= 2){
5       for (unsigned int j = k/2; j>0; j/=2){
6          unsigned int ixj = tid ^ j;
7          if( ixj > tid ) {
8             if( (tid & k)==0 ){…} else {…}
9          }
10         __syncthreads();
11      }
12   }
```

(a) The index value of a thread-ID (*tid*) solely determines whether the true path (*if*) or the false path (*else*) is taken. Hence, the branches at line 7 and 8 are programmatic.

Code #4) Branch dependent on *programmatic* values - (ii)

```
0    // Code snippet from the kernel of Mandelbrot benchmark
1    // imageW and imageH are scalar input parameters of the kernel
2
3    const int ix = blockDim.x * blockX + threadIdx.x;
4    const int iy = blockDim.y * blockY + threadIdx.y;
5    …
6    if( (ix < imageW) && (iy < imageH) )
7    {
8         …
9    }
```

(b) An intermediate variable (*ix, iy*), which is tainted by programmatic values, is combined with another programmatic value (*imageW, imageH*) from a scalar input parameter of the kernel. The branch at line 6 is therefore programmatic.

Figure 4.12: Example kernel codes containing P-branches.

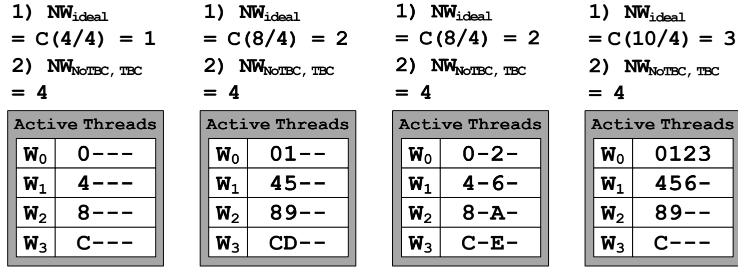
are all seen as constant values to all the threads within a CTA and are therefore programmatic values. In addition, the indexing value components of the thread-ID (e.g., `threadIdx.x, threadIdx.y`) are a virtual constant to the threads sharing the same index value (Figure 4.12(a)), and are also program-

matic. Compared to D-branches, branches depending on programmatic values (*P-branches*) are likely to be aligned because the (programmatic) values being used for resolving predicates are the same among threads (e.g., threads within the same warp or ones sharing a common home SIMD lane). Although P-branches that cause only partial alignment (Figure 4.13(c)) can be compacted as-is, this chapter shows that such cases are relatively rare compared to P-branches causing full alignment and preventing compaction (Figure 4.13(a,b)).

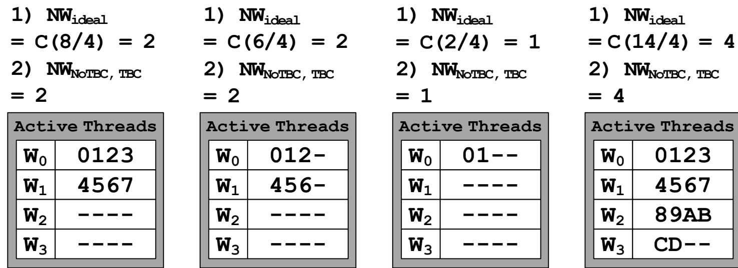
It is worth mentioning that a branch condition depending on both programmatic and data value (Figure 4.11(b)) behaves as a D-branch; this is because the data each thread is referencing will likely cause the predicate to be evaluated differently, regardless of the programmatic value. In Section 4.3.3 and Section 4.4.3.1, we categorize divergent branches into P-/D-branches and quantitatively verify our observations.

4.3.3 Programmatic Branches and Compactability

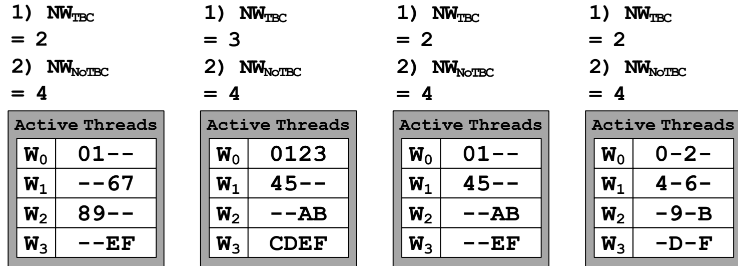
Figure 4.14 and Figure 4.15 summarizes the overall behavior of branch divergence and compactability of the studied benchmarks. First, Figure 4.14(a) shows a breakdown of all dynamically executed branches based on their divergence and potential compactability. The divergent branches are further categorized as P-/D-branches based on a taint analysis of the predicates of branch instructions using GPUOcelot [59] in Figure 4.14(b) (taint analysis is detailed in Section 4.4.1). In addition, we evaluate TBC_{ideal} and TBC's *compaction rate* across the divergent P-/D-branches in order to compare what fraction



(a) Programmatic value dependent control flows that are *potentially compactable* but not as-is.



(b) Programmatic value dependent control flows that are *never* compactable.

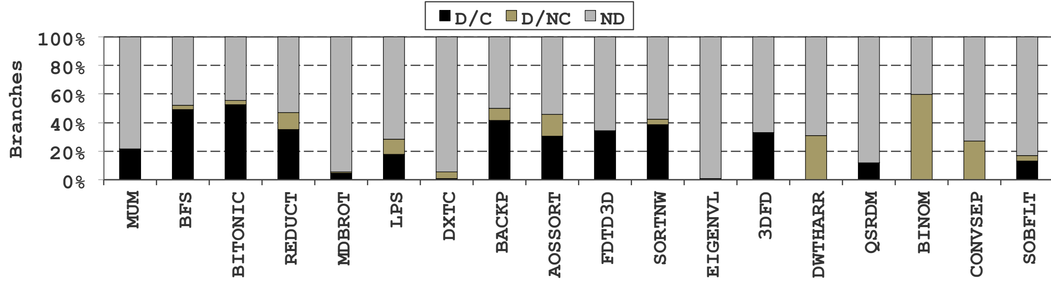


(c) Programmatic value dependent control flows that are compactable as-is.

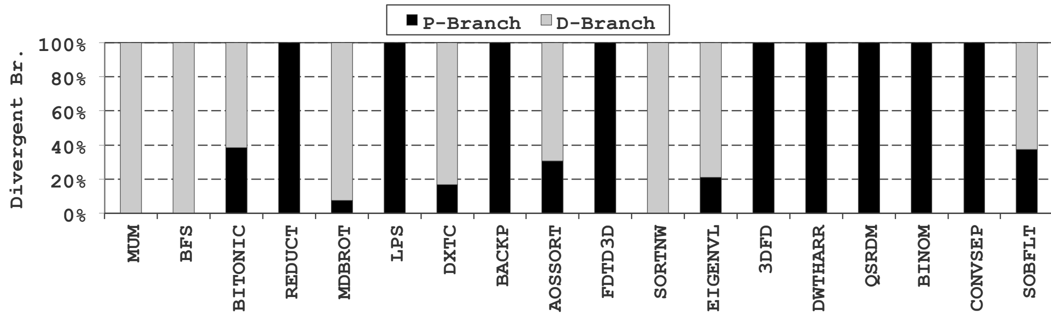
* C(X): Ceiled value of X

Figure 4.13: Active warp status for programmatic-value dependent control flow. W_n designates the warp with warp-ID n.

of the potential compaction opportunities are actually utilized (Figure 4.15). We define *compaction rate* as the fraction of compactable paths among all



(a) Breakdown of all dynamically executed conditional branches into: (i) non-divergent branches (ND), (ii) divergent branches with no potential compactability (D/NC), and (iii) divergent branches with potential compactability (D/C); potentially-compactable branches are those that can be compacted with TBC_{ideal}.



(b) Breakdown of divergent branches (D/NC and D/C in (a)) into P-branches and D-branches.

Figure 4.14: Categorization of branches based on control divergence, compactability, and P-/D-branch types.

the (CTA-wide) paths generated from divergent branches (true/false) in an application.

To quantify our results, we evaluate the number of warps for each path when applying compaction with TBC or ideally. We use NW_{NoTBC} to refer to the number of warps in a CTA without compaction applied, NW_{TBC} for the number of warps after compaction, and NW_{ideal} for the minimum number of warps possible with any compaction mechanism. The lower bound on

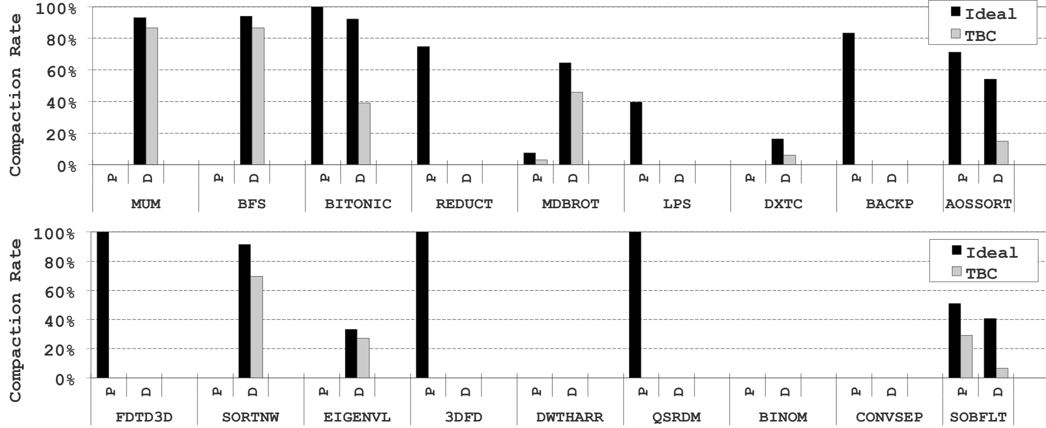


Figure 4.15: Compaction rate of P-branches and D-branches using TBC_{ideal} and TBC. Note that because all divergent branches of DWTHARR, BINOM, and CONVSEP are never compactable (D/NC), the corresponding TBC_{ideal} compaction rates are zero.

compaction is simply the minimum number of warps needed to execute the number of threads in each path of the CTA:

$$NW_{ideal} = \lceil \frac{NumActive_{CTA}}{SIMD_{width}} \rceil \tag{4.1}$$

where, $NumActive_{CTA}$ refers to the total number of threads active at the diverging path across the CTA and $SIMD_{width}$ designates the width of the SIMD pipeline. When NW_{ideal} and NW_{TBC} (number of warps after compaction with TBC) for a path are both smaller than the number of warps without compaction (NW_{NoTBC}), then the path can be compacted as-is with TBC. When NW_{NoTBC} and NW_{TBC} are equal and NW_{ideal} is smaller, a path is potentially compactable but cannot be compacted as-is with TBC (Figure 4.13(a)). When

the values of NW_{ideal} and NW_{NoTBC} are the same, a path has neither potential nor actual compactability (Figure 4.13(b)).

By definition, benchmarks with a nonzero TBC compaction rate in either P-/D-branches (Figure 4.14(c)) exhibit improvements in $SIMD_{util}$ with TBC. As expected, all 9 benchmarks containing any D-branches exhibit nonzero TBC compaction rates for this branch type, and we confirm our intuition that previous compaction mechanisms work relatively well for D-branches. However, in 13 of the 15 benchmarks containing P-branches, no compaction occurs at all, as seen by their zero P-branch TBC compaction rate in Figure 4.15. Of these 13 benchmarks, only 5 have no potential for compaction (zero compaction with TBC_{ideal}). The other 8 benchmarks (BITONIC, REDUCT, LPS, BACKP, AOSSORT, FDTD3D, 3DFD, and QSRDM) show that substantial opportunity for improving $SIMD_{util}$ is untapped with TBC and other current compaction techniques because a thread's home SIMD lane is fixed. My goal is to tackle such P-branches and enable new compaction opportunities using SIMD lane permutation, which is described in the following section.

4.3.4 Motivation and Key Insights

SIMD lane permutation (SLP) [26] is based on the insight that if the home SIMD lanes of threads are permuted from their sequentially-assigned locations, the alignment of threads can be eliminated in many cases. The permutation is applied to each thread when a warp is launched, before it starts executing, and is then fixed until the warp completes execution. This

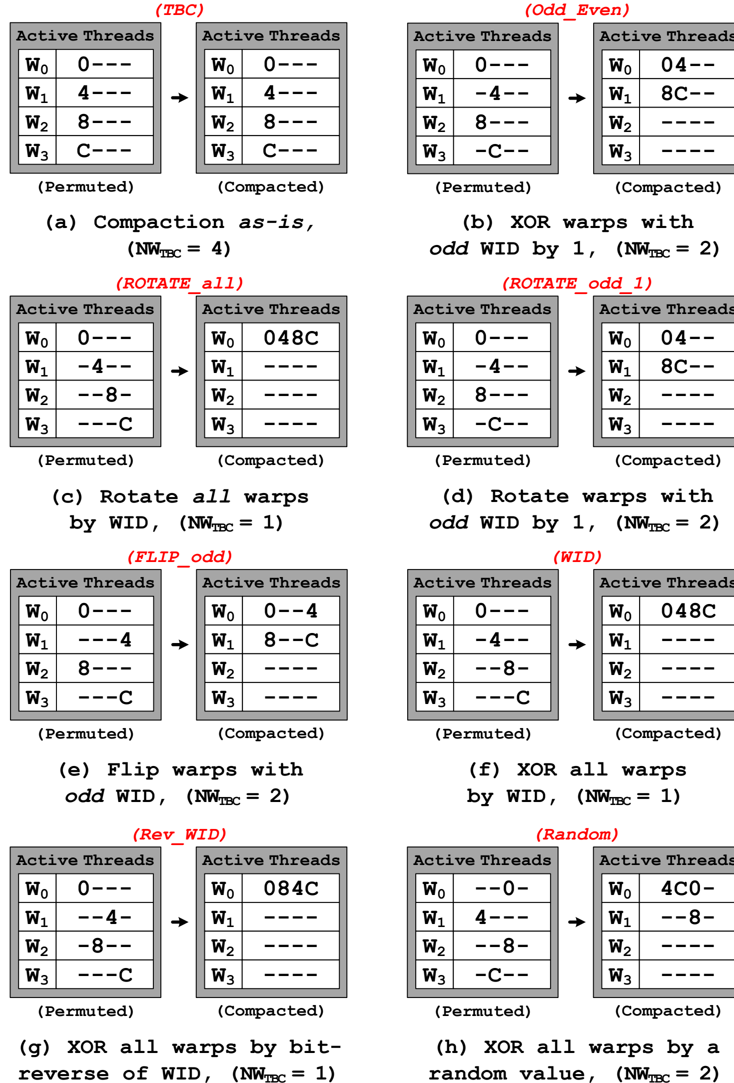


Figure 4.16: Examples of various SIMD lane permutation mechanisms that alter the alignment of active threads to lanes.

improves compactability while maintaining the cost-efficiency of the baseline compaction mechanism and SIMT architecture[5].

Permuting the home SIMD lane of each thread requires changing how each thread-ID is mapped to a SIMD lane. Figure 4.16 illustrates some example permutations and their associated mapping, including the ones discussed in prior work [44, 22]. Because these mapping functions can be calculated statically using only the thread-IDs and warp size, the compaction-hardware requires no modification. While older NVIDIA GPUs mandated that threads access memory in sequence to enable memory coalescing and minimize memory transactions, AMD GPUs and recent NVIDIA GPUs (starting with NVIDIA Compute Capability 2.0) can coalesce any collection of addresses within a warp into the minimum possible number of transactions [28]. Thus, thread order within a warp does not impact performance and SLP can be incorporated smoothly without disrupting memory access behavior.

Note that SLP can impact legacy codes that are highly optimized by expert programmers, leveraging undocumented behavior of SIMT programming. For instance, if two threads within a warp write to the same memory location, it is undefined which thread will succeed under the CUDA programming model [28]. In practice, for a given hardware implementation, it is predictable

[5]Fung et al. [22] conducted a preliminary study on the potential of permutation for better compaction. The authors adopted a simple permutation to mitigate lane conflicts for compaction on BITONIC (Table 4.2), but this thesis demonstrates cases where this permutation falls short.

```
Code #5) Programmatic branch causing only the 1st half of the warp active
0    // Code snippet from the kernel of BACKP benchmark
1    // CTA is a (8 × 16) 2-D array of threads.
2
3    int tx = threadIdx.x;
4    int ty = threadIdx.y;
5    …
6    for (int i=1; i<=__log2f(HEIGHT); i++){
7      int power_two = __powf(2,i);
8
9      if( ty % power_two == 0 ) {…}
10     …
11   }
```

Figure 4.17: Code snippet from BACKP benchmark that exhibits programmatic divergence. The resulting aligned divergence is illustrated in Figure 4.18(a).

and consistent which expert programmers utilize to tune the kernel. We detail SLP's impact on programmability in Section 4.5.2.

4.3.5 Pitfalls of a Random Permutation

While the permutations in Figure 4.16(b)-(h) can all break the alignment of active lanes in some cases, their effectiveness in increasing compactability can vary substantially because some permutations are only optimal for certain divergence patterns. *Odd_Even* [22], for example, works most effectively for the alignment pattern shown in Figure 4.18(b), but provides no benefit when active threads are grouped together as in Figure 4.18(a) (active/active, inactive/inactive). Another permutation, such as *XOR*-ing only

	Lane-ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
For W0	TID	0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0	0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1
For W1	TID	0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2	0,3	1,3	2,3	3,3	4,3	5,3	6,3	7,3
For W2	TID	**0,4**	**1,4**	**2,4**	**3,4**	**4,4**	**5,4**	**6,4**	**7,4**	0,5	1,5	2,5	3,5	4,5	5,5	6,5	7,5
For W3	TID	0,6	1,6	2,6	3,6	4,6	5,6	6,6	7,6	0,7	1,7	2,7	3,7	4,7	5,7	6,7	7,7
For W4	TID	**0,8**	**1,8**	**2,8**	**3,8**	**4,8**	**5,8**	**6,8**	**7,8**	0,9	1,9	2,9	3,9	4,9	5,9	6,9	7,9
For W5	TID	0,A	1,A	2,A	3,A	4,A	5,A	6,A	7,A	0,B	1,B	2,B	3,B	4,B	5,B	6,B	7,B
For W6	TID	**0,C**	**1,C**	**2,C**	**3,C**	**4,C**	**5,C**	**6,C**	**7,C**	0,D	1,D	2,D	3,D	4,D	5,D	6,D	7,D
For W7	TID	0,E	1,E	2,E	3,E	4,E	5,E	6,E	7,E	0,F	1,F	2,F	3,F	4,F	5,F	6,F	7,F

Threads that are active when '*power_two*' is '4'

	Lane-ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
For W0	TID	**0,0**	**1,0**	**2,0**	**3,0**	**4,0**	**5,0**	**6,0**	**7,0**	0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1
For W1	TID	**0,2**	**1,2**	**2,2**	**3,2**	**4,2**	**5,2**	**6,2**	**7,2**	0,3	1,3	2,3	3,3	4,3	5,3	6,3	7,3
For W2	TID	**0,4**	**1,4**	**2,4**	**3,4**	**4,4**	**5,4**	**6,4**	**7,4**	0,5	1,5	2,5	3,5	4,5	5,5	6,5	7,5
For W3	TID	**0,6**	**1,6**	**2,6**	**3,6**	**4,6**	**5,6**	**6,6**	**7,6**	0,7	1,7	2,7	3,7	4,7	5,7	6,7	7,7
For W4	TID	**0,8**	**1,8**	**2,8**	**3,8**	**4,8**	**5,8**	**6,8**	**7,8**	0,9	1,9	2,9	3,9	4,9	5,9	6,9	7,9
For W5	TID	**0,A**	**1,A**	**2,A**	**3,A**	**4,A**	**5,A**	**6,A**	**7,A**	0,B	1,B	2,B	3,B	4,B	5,B	6,B	7,B
For W6	TID	**0,C**	**1,C**	**2,C**	**3,C**	**4,C**	**5,C**	**6,C**	**7,C**	0,D	1,D	2,D	3,D	4,D	5,D	6,D	7,D
For W7	TID	**0,E**	**1,E**	**2,E**	**3,E**	**4,E**	**5,E**	**6,E**	**7,E**	0,F	1,F	2,F	3,F	4,F	5,F	6,F	7,F

Threads that are active when '*power_two*' is '2'

*** Each CTA is a (8 × 16) array of threads.**
*** Lane-ID = (threadIdx.y*8 + threadIdx.x) % ($SIMD_{width}$)**
*** $SIMD_{width}$ and warp size are both 16.**

(a) Active threads with value-dependent lane alignment, corresponding to the branch at line 9 of Figure 4.17 (dependent on *power_two*).

	Lane-ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
For W0	TID	**0**	1	**2**	3	**4**	5	**6**	7	**8**	9	**10**	11	**12**	13	**14**	15
For W1	TID	**16**	17	**18**	19	**20**	21	**22**	23	**24**	25	**26**	27	**28**	29	**30**	31
For W2	TID	**32**	33	**34**	35	**36**	37	**38**	39	**40**	41	**42**	43	**44**	45	**46**	47
For W3	TID	**48**	49	**50**	51	**52**	53	**54**	55	**56**	57	**58**	59	**60**	61	**62**	63

Threads that are active when '(j == 1)', having *((ixj > tid) == true)*

*** Each CTA is a (64 × 1) array of threads.**
*** Lane-ID = (threadIdx.x) % ($SIMD_{width}$)**

(b) Active threads with programmatic lane alignment, corresponding to line 7 of Figure 4.12(a)

Figure 4.18: Examples of aligned divergence patterns in Figure 4.19(b–c).

the odd-ID warps with $\left(\frac{SIMD_{width}}{2}\right)$, on the other hand, will perfectly compact all the threads in Figure 4.18(a), but none for Figure 4.18(b).

Incorporating randomness in the permutation function (e.g., *Random*, *WID*, and *Rev_WID* in Figure 4.16(f–h)) intuitively seem more effective in

redistributing clustered threads. However, such random permutations do not perform well when a large fraction of the threads are active across the CTA. For example, consider a case where half the threads in each warp are active – unless the active threads are permuted exactly to the other half of the (vacant) lanes, compaction will fail. Figure 4.18 shows an example of a P-branch causing half of the threads in a CTA to be active. This behavior is exhibited in applications such as BITONIC, QSRDM, MDBROT, and others. While such a divergence pattern can occur from an under-optimized kernel, it can also be generated by the nature of the algorithm itself. Note that *Odd_Even* and *Rev_WID* both only effectively compact one of the two example patterns but not both (Figure 4.19). In general, I observe that previously discussed permutation functions fall short of ideal and are not robust because they are based on empirical observations of a subset of divergence patterns.

4.3.6 Balanced Permutation

Based on the previous discussion, this chapter presents a robust permutation mechanism. An important insight is that the permutation to be applied should distribute active threads across the SIMD lanes in a *balanced* manner because aligned divergence from P-branches frequently exhibits highly skewed distributions of active lanes. *Balanced* is designed such that for any CTA (with fewer than $SIMD_{width}$ warps) each physical lane only has a single instance of each logical thread location within a warp. Among all previously discussed permutation mechanisms [22, 44], this characteristic is unique to

XOR-ed values to even/odd-ID warp pairs, when XOR-ed with each other are always equal to a full-mask with bitlength $\log_2(SIMD_{width})$, which is 111 in this example.

All odd-ID warps are XOR-ed with values larger than the $(SIMD_{width}/2)$, which effectively swaps the 1st half of the lanes with the 2nd half.

Permuted lanes are always perfectly balanced within each physical lane.

	Lane-ID	0 (000)	1 (001)	2 (010)	3 (011)	4 (100)	5 (101)	6 (110)	7 (111)	(Function of Permutation)
For W0	XOR-000	0	1	2	3	4	5	6	7	: Maintain as-is
For W1	XOR-111	7	6	5	4	3	2	1	0	: Flip end-to-end
For W2	XOR-001	1	0	3	2	5	4	7	6	: Permute even-odd lanes
For W3	XOR-110	6	7	4	5	2	3	0	1	: Swap half and *XOR-010*
For W4	XOR-010	2	3	0	1	6	7	4	5	: Swap within each half warp
For W5	XOR-101	5	4	7	6	1	0	3	2	: Swap half and *XOR-001*
For W6	XOR-011	3	2	1	0	7	6	5	4	: Flip within each half
For W7	XOR-100	4	5	6	7	0	1	2	3	: Swap 1st half with 2nd half
⋮	Repeat above									

(a) Proposed *(Balanced)* Algorithm (assuming WarpSize and $SIMD_{width}$ of 8).

Active Threads	*(TBC)*	*(Odd_Even)*	*(Rev_WID)*	*(Balanced)*
W_0	0-2-4-6-	0-2-4-6-	0-2-4-6-	0-2-4-6-
W_1	8-A-C-E-	-8-A-C-E	C-E-8-A-	-E-C-A-8
W_2	G-I-K-M-	G-I-K-M-	I-G-M-K-	-G-I-K-M
W_3	O-Q-S-U-	-O-Q-S-U	U-S-Q-O-	U-S-Q-O-

(b) Example control flow where *(Rev_WID)* does not compact well.

Active Threads	*(TBC)*	*(Odd_Even)*	*(Rev_WID)*	*(Balanced)*
W_0	0123----	0123----	0123----	0123----
W_1	89AB----	98BA----	----89AB	----BA98
W_2	GHIJ----	GHIJ----	IJGH----	HGJI----
W_3	OPQR----	PORQ----	----QROP	----QROP

(c) Example control flow where *(Odd_Even)* does not compact well.

Figure 4.19: Proposed *Balanced* permutation.

Balanced and provides its robustness. Intuitively, threads within warps with an even warp-ID (WID) are only permuted within each half-warp (each warp uses a different *XOR* mask). Every even-ID warp is paired with an odd-ID warp that uses a complementary mask and ensures that threads are shuffled

to the second half of each warp. The permutation algorithm is to *XOR* the logical thread location with a mask that is computed differently for each warp using the warp-ID (WID). The *Balanced* permutation masks are computed for even warp-IDs using the formula in Equation 4.2. The masks for odd warp-IDs are a bit-wise inverse of their even warp-ID pairs.

$$XOR_{evenWID} = \frac{evenWID}{2} \tag{4.2}$$

Figure 4.19 illustrates the functionality of SLP with the proposed *Balanced* permutation algorithm. Without SLP, lane-ID 0 is always assigned to physical lane 0, for example. With *Balanced*, on the other hand, each physical lane only has a single instance of each lane-ID. This *vertical balance* is unique to the carefully-designed *Balanced* permutation and is clearly shown in the highlighted physical lane 7 in Figure 4.19. Figure 4.19 also illustrates the overall construction of *Balanced*. Randomized permutations only achieve this balance on average, while any individual CTA is likely to be somewhat imbalanced. This imbalance is greater on average for CTAs that have a small number of warps. *Balanced* works well even for these CTAs, because of the even/odd complementary masks it uses.

This chapter presents a detailed quantitative evaluation of various permutations in Section 4.4.3, but in summary, *Balanced* is very robust and is either the most effective permutation, or is within 99.4% of the best permutation in all our experiments. Because *Balanced* tends to outperform other

permutations, is within a small fraction of the best permutation in the rare cases it is not already the best, and rarely degrades performance, I argue that *Balanced* should always be applied when allocating a new warp on compaction-based GPU architectures.

4.3.7 Summary of the Benefits of SLP

Previous SIMT compaction mechanisms, even with CAPRI enabled, fall short because of their limited applicability. As we explore in Section 4.4.3, the way threads are associated with SIMD lanes causes aligned divergence patterns that prevent compaction, which mainly originates from non-data dependent, programmatic branches. Although diverging paths from these programmatic branches do not compact well as-is, there is substantial opportunity if the fixed association of threads to lanes can be relaxed. The proposed SLP expands the applicability of compaction by reducing, and even eliminating, aligned divergence. SLP permutes the mapping of logical thread locations to physical SIMD lanes when a warp is launched. This breaks aligned divergence patterns resulting from conditionals that depend only on programmatic values. The novel and robust *Balanced* permutation technique enables significant improvements in compacting programmatic branches, widening the applicability of compaction.

Table 4.1: Simulator configuration for CAPRI/SLP evaluation.

Number of GPU cores	30
Threads per GPU core	1024
Threads per warp	32
SIMD lane width	32
Registers per GPU core	16384
Shared memory per GPU core	32KB
Warp scheduling policy	Two-level round-robin [24]
L1 Cache (size/associativity/block size)	32KB/8-way/64B
L2 Cache (size/associativity/block size)	1024KB/64-way/64B
Memory bandwidth	102.4 GB/s
Memory controller	Out-of-order (FR-FCFS)

4.4 Evaluation

This section first describes the evaluation methodology followed by a detailed evaluation of CAPRI (Section 4.4.2) and SLP (Section 4.4.3).

4.4.1 Methodology

The microarchitectural components of CAPRI and SLP are modeled using GPGPU-Sim [47], a detailed cycle-level performance simulator of a "general purpose" GPU architecture. TBC as well as TBC+ (a version of TBC that does not stall warps from branches that cannot diverge as indicated by the compiler) are also implemented. We configure the simulator to closely match NVIDIA's Quadro FX5800 as detailed in the GPGPU-Sim manual (see Table 4.1). The two-level round-robin warp scheduling policy [24] is used for issuing warps to execute. This scheme divides all active warps within a GPU core into multiple groups of warps and chooses which group to preferentially

```
Pseudo PTX Program) Taint analysis

0    …
1    mov        %r2, %tid.x    // %r2 is tainted by a programmatic value
2    setp.eq    %p1, %r2, 0    // Predicate register %p1 tainted from %r2
3    @%p1 bra   [JUMP_1]       // Branch depends on programmatic value
4    ld.param   %r3, [g_data]  // Pointer to a data array is loaded to %r3
5    add        %r4, %r2, %r3  // %r4 contains the address to load from
5    ld.global  %r5, [%r4]     // %r5 is tainted by data array value
6    setp.gt    %p6, %r5, 1    // Predicate register %p6 tainted from %r5
7    @%p6 bra   [JUMP_2]       // Branch depends on data value
8    mov        %r2, 100       // Taint is cleared by an immediate value
9    …

* tid.x   : x-index value of a thread-ID
* g_data  : data array allocated at global memory
```

Figure 4.20: Pseudo PTX code explaining the taint analysis methodology.

schedule in a round-robin manner. The warps in the currently prioritized group maintain the highest scheduling priority until all of that group's warps are stalled; in which case the next group is prioritized in scheduling. This study configures the number of warps within a prioritized group to match the number of warps that compose a single CTA.

CAPRI. We use a 32-entry CAPT with a 1-bit latest history configuration for our default CAPRI configuration. As mentioned in Section 4.2.2, a 2-bit saturating counter and a 1-bit sticky adequacy configuration are also evaluated and we explicitly note when using different parameters to evaluate CAPRI sensitivity.

SLP. The effectiveness of SLP is evaluated using a combination of GPUOcelot [59] and GPGPU-Sim. GPUOcelot is an open source compiler infrastruc-

ture supporting NVIDIA's PTX version 3.0; we use its PTX emulator to classify divergent branches into P-/D-branches using taint analysis (Figure 4.20). We also leverage the CTA-wide active mask information in GPUOcelot to analyze compaction rate (Section 4.4.3.1) and $SIMD_{util}$.

Benchmarks. CAPRI and SLP have been studied with 40 benchmarks from CUDA-SDK [3], Rodinia [14], and those provided with GPGPU-Sim [47, 51, 60, 61, 62, 63]. Of these 40 benchmarks, 18 exhibit branch divergence (the other 22 have $SIMD_{util}$ of over 99%), and 8 exhibit increased idle cycles due to the synchronization overhead of TBC. This chapter therefore primarily focuses on the 8 benchmarks when discussing the benefits of CAPRI and the 18 benchmarks for SLP evaluations[6]. The benchmarks are summarized in Table 4.2.

4.4.2 CAPRI Results and Analysis

This section provides a detailed evaluation of CAPRI including the quality of the predictions, its impact on SIMD lane utilization and idle cycles, the performance improvements it provides, parameter sensitivity, and implementation cost.

---

[6]Note that GPGPU-Sim does not support some runtime APIs, such as OpenGL, which are supported by GPUOcelot. We therefore only report the performance results of the 8 out of 18 benchmarks that GPGPU-Sim can simulate without modification. All CUDA applications were compiled as-is and with the parameters provided with GPGPU-Sim and GPUOcelot.

Table 4.2: Benchmarks studied for CAPRI/SLP evaluation.

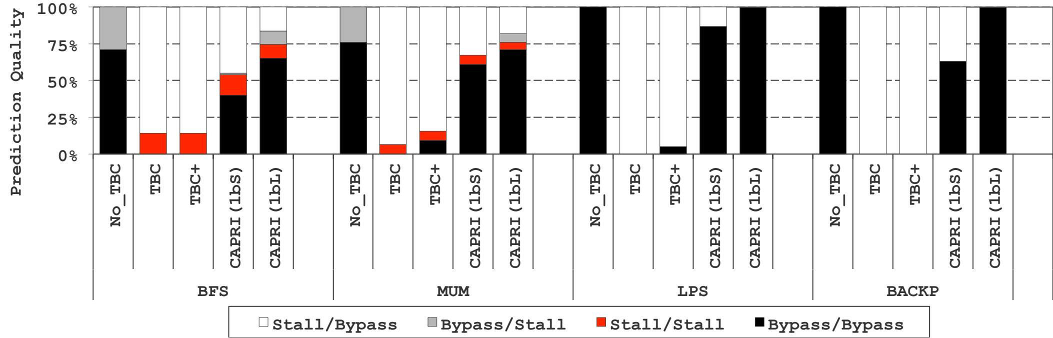
Abbreviation	**Description**	**#Instr.**	**Ref.**
LPS	3D laplace solver	985K	[47]
BFS	Breadth first search	256K	[47]
MUM	MUMmerGPU	2.7M	[47]
LIB	LIBOR monte carlo	1B	[47]
BACKP	Back propagation	1M	[14]
PFLT	Particle filter	4B	[14]
BITONIC	Bitonic sort	2K	[3]
REDUCT	Reduction (Kernel 0)	44K	[3]
MDBROT	Mandelbrot	8.3M	[3]
DXTC	DXT compression	955K	[3]
AOSSORT	AOS sorting	39K	[3]
FDTD3D	FDTD stencil on 3D	71M	[3]
SORTNW	Sorting network	2M	[3]
EIGENVL	Eigen-value	6.7M	[3]
3DFD	Finite diff. comp. 3D	29K	[3]
DWTHARR	Harr wavelets	2K	[3]
QSRDM	Quasirandom generator	3.1M	[3]
BINOM	Binomial options	176K	[3]
CONVSEP	Separable convolution	204K	[3]
SOBFLT	Sobel filter	575K	[3]

4.4.2.1 Prediction Quality

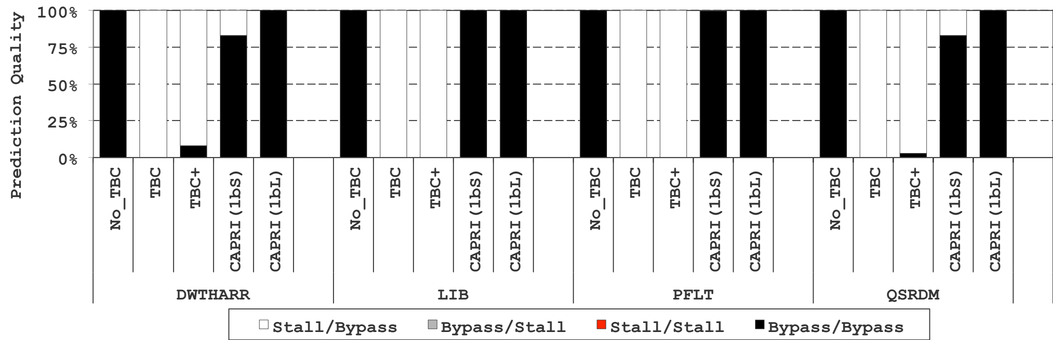
CAPRI predicts whether the warps in a given branch should stall and synchronize to attempt compaction or whether they should continue to execute without compaction. CAPRI can thus correctly or incorrectly predict to stall or to bypass. Figure 4.21 shows the quality of the predictions by categorizing each prediction as correctly predicting to bypass and not wait for compaction (Bypass/Bypass), correctly predicting to stall and wait for compaction (Stall/Stall), incorrectly predicting to bypass while compaction would have been effective (Bypass/Stall), and incorrectly predicting to wait when

compaction is ineffective (Stall/Bypass). We determine which category each prediction falls into in the following way. For each branch, we perform an exhaustive search of all potential compactions and identify the minimum number of warps required to achieve maximal compaction for each dynamic branch. We then count how many warps were bypassed and how many waited at each specific dynamic branch.

Baseline without compaction (*No_TBC*) and TBC "predict" that all warps should always bypass or always stall, respectively. TBC, for example, only generates correct Stall/Stall or, incorrect, Stall/Bypass decisions. Because most branches are not adequate candidates for compaction (recall examples in Section 4.2), most decisions made by TBC are to incorrectly stall a warp when it should have just continued to execute without waiting for compaction. TBC makes the smallest fraction of correct decisions between all the schemes. In fact, TBC has near-zero accuracy with non-divergent workloads, which explains why it degrades performance in some cases. *No_TBC* makes the opposite decisions to TBC, either correct Bypass/Bypass or incorrect Bypass/Stall. *No_TBC* makes near-perfect decisions for non-divergent workloads. In the divergent cases, *No_TBC* still makes correct decisions for roughly 50% of warps. It is interesting to observe that *No_TBC* has near perfect accuracy in LPS, BACKP, and DWTHARR, while TBC and TBC+ make nearly no correct predictions. These three benchmarks exhibit branch divergence, but compaction is ineffective (see also Section 4.4.2.2).



(a) Divergent benchmarks



(b) Non-divergent benchmarks

Figure 4.21: Prediction and stall/bypass decision quality. Each bar represents the fraction of warps that were correctly bypassed or stalled (Bypass/Bypass or Stalled/Stalled), and incorrectly bypassed or stalled (Bypass/Stall or Stall/Bypass). In this section, we categorize benchmarks as being *divergent* when its average SIMD lane utilization is below 90% and *non-divergent* otherwise.

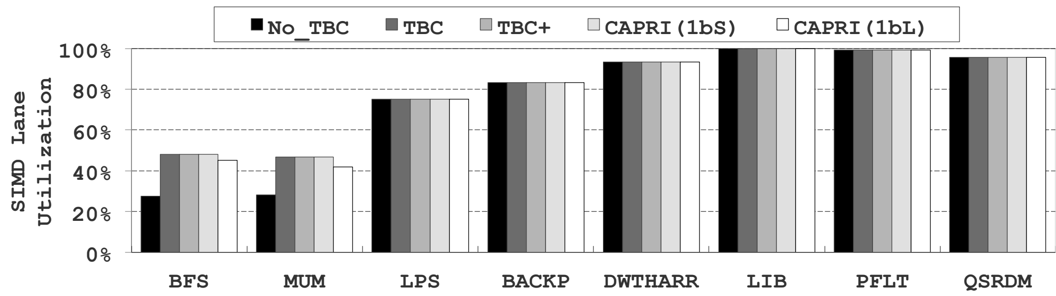
Unlike TBC, TBC+ decides to bypass warps at branch points when it is statically known that no divergence can occur. There is only a small number of these branches, however, and over 80% of the decisions made by TBC+ are still incorrect. CAPRI, on the other hand, makes high-quality decisions in nearly all cases. CAPRI accurately predicts 99% and 87% of the

warps ideal behavior on non-divergent and divergent benchmarks, respectively. BFS and MUM have a highly irregular divergence pattern that is hard to predict. CAPRI still achieves an average of 61% and 75% accuracy using the 1b-Sticky and 1b-Latest history bit configurations, respectively. As expected, 1b-Sticky has overall lower accuracy, but makes more correct Stall/Stall predictions because of its bias, compared to 1b-Latest that may incorrectly react to anomalous dynamic behavior. Performance with 1b-Latest is higher, indicating that predictions should not be biased towards stalling. The correct decisions made in these three benchmarks result in highly-effective compaction (see Section 4.4.2.2). As a result, the number of dynamic warps is much smaller after compaction than in the baseline *No_TBC*, which is why the fraction of stall decisions is relatively low even though *No_TBC* has a large fraction of incorrect Bypass/Stall decisions.

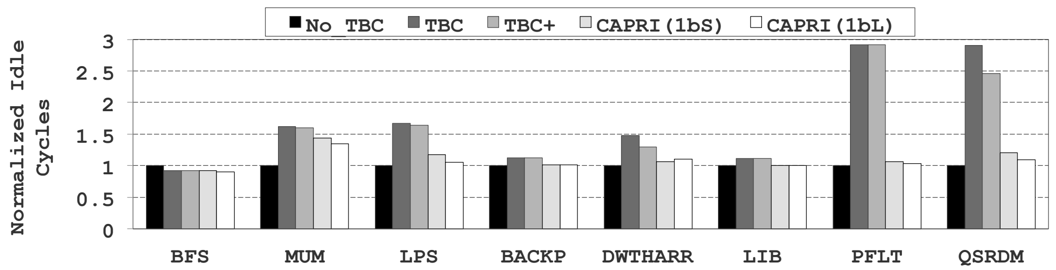
In addition to the configurations discussed above, a 2-bit saturating counter based predictor was also studied. Accuracy and performance improved marginally, by less than 1%, and the results have been omitted for brevity.

4.4.2.2 SIMD Lane Utilization and Idle Cycles

Figure 4.22(a) and (b) show the impact of compaction on SIMD lane utilization and the number of idle cycles. Overall, TBC and TBC+ significantly improve SIMD lane utilization, by up to a factor of 2 for BFS and MUM. CAPRI is able to correctly predict beneficial compaction and matches the SIMD utilization improvements of TBC, to within 100% and 98.8% for



(a) SIMD lane utilization.



(b) Normalized idle cycles accumulated across all GPU cores.

Figure 4.22: SIMD lane utilization and (normalized) idle cycle count.

the two history configurations: The 1b-Sticky history stalls a larger number of threads but achieves marginally better compaction than with a 1b-Latest configuration.

TBC does not achieve improvements for all other benchmarks, even for LPS and BACKP that have significant divergence. This indicates that compaction is ineffective and will not improve performance, while stalling warps may degrade performance. On average, TBC increases idle cycles by an average 72% across the 8 applications. By avoiding stalls on unconditional branches, TBC+ introduces fewer idle cycle, but still has 63% more idle cycles than the baseline *No_TBC* on average. While CAPRI matches the gains of

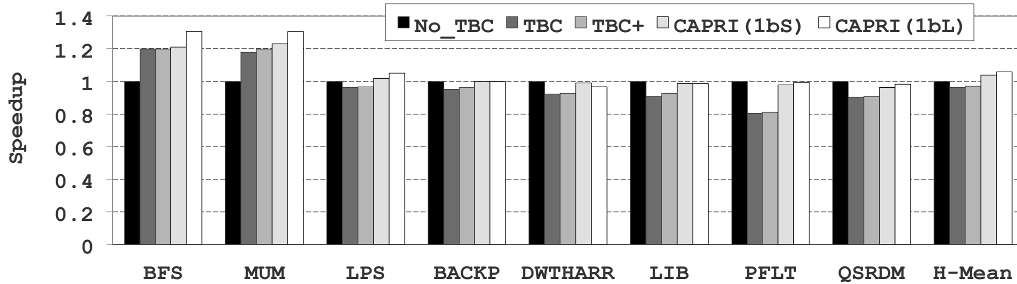
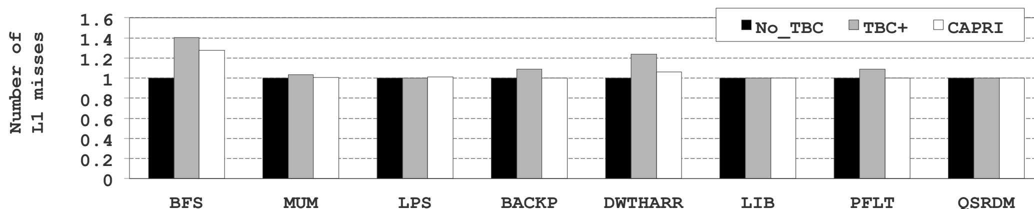


Figure 4.23: Performance of CAPRI compared to TBC and TBC+.

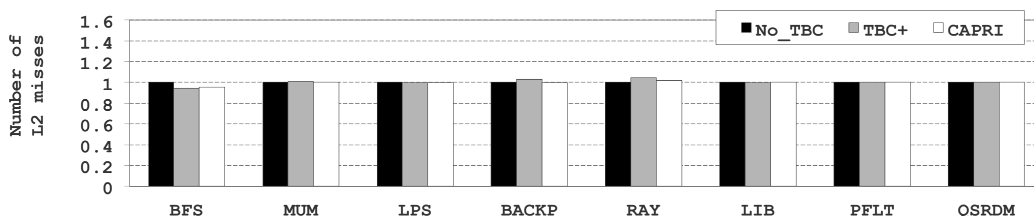
TBC, it does much better with respect to idle cycles. By not stalling most warps that do not benefit from compaction, CAPRI does not increase idle cycles by more than 40% for *any* application and incurs an average of only 7% and 5% increase for the 1b-Sticky and 1b-Latest history configurations respectively. Based on these results we expect 1b-Sticky to outperform all other schemes because it is nearly optimal in terms of compaction gains and has the smallest increase of idle cycles.

4.4.2.3 Overall Performance

Figure 4.23 shows the performance of CAPRI relative to that of *No_TBC*, TBC, and TBC+. CAPRI outperforms TBC and TBC+ on all benchmarks because it generally correctly distinguishes between warps that can benefit from compaction and those that only suffer unnecessary synchronization delays. Concerning divergent workloads, CAPRI with 1b-Latest performs 14.9% and 7.6% on average (harmonic mean) better than *No_TBC* and TBC+, respectively. As expected, the 1b-Sticky configuration does not perform as well as 1b-Latest, but still outperforms *No_TBC* and TBC+. In terms of non-



(a) L1 misses.



(b) L2 misses.

Figure 4.24: Changes in L1 and L2 miss count from compaction, normalized to *No_TBC*. CAPRI is configured with 1b-Latest.

divergent benchmarks, CAPRI dynamically evaluates non-compactability and allows warps to bypass compaction barriers, thereby minimizing performance loss. Note that TBC and TBC+ degrade the performance of the 6 benchmarks, excluding BFS and MUM, because of the excessive synchronization overheads with no gains in SIMD lane utilization. For non-divergent workloads, accordingly, TBC and TBC+ suffer an average of 12% and 11% performance degradation respectively.

4.4.2.4 Impact on the Memory System

Compacting warps involves rearranging threads from the CTA. Because different groups of threads execute concurrently compared to program order and the baseline *No_TBC*, it is possible that the application memory behavior

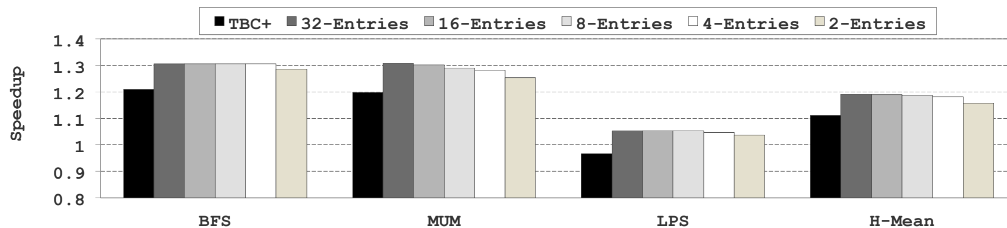


Figure 4.25: Performance of CAPRI (1b-Latest) with variable CAPT size.

changes as well. Figure 4.24 shows the relative number of first- and second-level cache misses for TBC+ and CAPRI normalized to *No_TBC*. TBC+ significantly increases the number of L1 misses for certain applications such as BFS, MUM, BACKP, and DWTHARR. The reason for this change in non-divergent applications is that the WCU rearranges threads even when compaction is ineffective (number of warps is not reduced), which can aggravate memory divergence and increase cache miss rate (see Figure 4.6). CAPRI, on the other hand, does much better than TBC+. CAPRI correctly predicts the lack of compaction-effectiveness for many branches, including nearly all branches in the non-divergent workloads. As a result, misses do not increase for these workloads and are reduced (compared to TBC+) for the highly-divergent applications. Note that for BFS and MUM, which exhibit noticeable increase in L1 cache misses, performance is actually most improved because of the large benefits from increased SIMD utilization through compaction. Even though L1 misses have increased, the missing working sets at L1 are still mostly resident at shared L2 caches, maintaining L2 caching efficiency and memory traffic.

Table 4.3: CAPRI area overheads.

Number of NAND gate counts required for CAPT				
2-entry	**4-entry**	**8-entry**	**16-entry**	**32-entry**
733.56	1432.63	2840.43	6005.44	12338.07

4.4.2.5 Sensitivity Studies

Figure 4.25 shows the normalized IPC achieved when varying the number of CAPT entries on a subset of the divergent benchmarks. We only discuss those benchmarks for which the number of entries has noticeable impact. All other benchmarks achieved at least 98.3% of the IPC of an unlimited CAPT with just 4 entries – mainly because non-divergent branches are naturally bypassed with CAPRI's algorithm. MUM has the most sensitivity to the number of entries used. This application achieves over 95% of the IPC benefits of an unbounded CAPT with just 4 entries and 86% with a 2-entry CAPT. All results use LRU replacement and a fully-associative organization.

4.4.2.6 Implementation and Energy-Efficiency of CAPRI

Fung and Aamodt [23] evaluated the overhead of TBC and showed it to be less than $1mm^2$ for an entire chip ($500mm^2$) in $65nm$ technology. In addition to this small area, CAPRI requires area for the CAPT and for adequacy evaluation in the WCU. The adequacy evaluator (predictor) added to the WCU can be implemented using simple logic for counting the number of ones in each SIMD lane, which represent an active thread in that lane. This logic is significantly simpler than the compaction logic itself and should

add negligible area to the WCU. As previously discussed, an 8-entry CAPT is sufficient to attain near-maximal performance improvement. Regardless, we quantify the implementation and energy overhead of a 32-entry CAPRI below.

The key microarchitectural components of CAPRI have been implemented using Verilog HDL [64], and Table 4.3 shows the implementation overhead of CAPT in terms of its overall gate count. The Synopsys Design Compiler and a 45nm high performance CMOS standard cell frontend library are used to synthesize the HDL codes into the gate level netlist. To evaluate energy, a trace of each benchmark's read/write accesses to the CAPT has been extracted from a single GPU core on a cycle-by-cycle basis using GPGPU-Sim. The switching activity of the CAPT is determined by running the traces through Synopsys Primetime-PX for power analysis. Overall, CAPRI with a 32-entry CAPT requires only 12K NAND gates and consumes less than 10mW, per GPU core. The implementation overhead of CAPRI is therefore expected to be negligible both in area and power.

4.4.3 SLP Results and Analysis

This section evaluates SLP (on top of CAPRI) in terms of the enhancements in compactability, SIMD lane utilization, and performance. TBC is assumed as the baseline compaction mechanism when discussing compactability (Section 4.4.3.1) and SIMD lane utilization (Section 4.4.3.2), so that all compactable branches are considered. For performance evaluations (Section 4.4.3.3), on the other hand, TBC is augmented with CAPRI as it consistently outper-

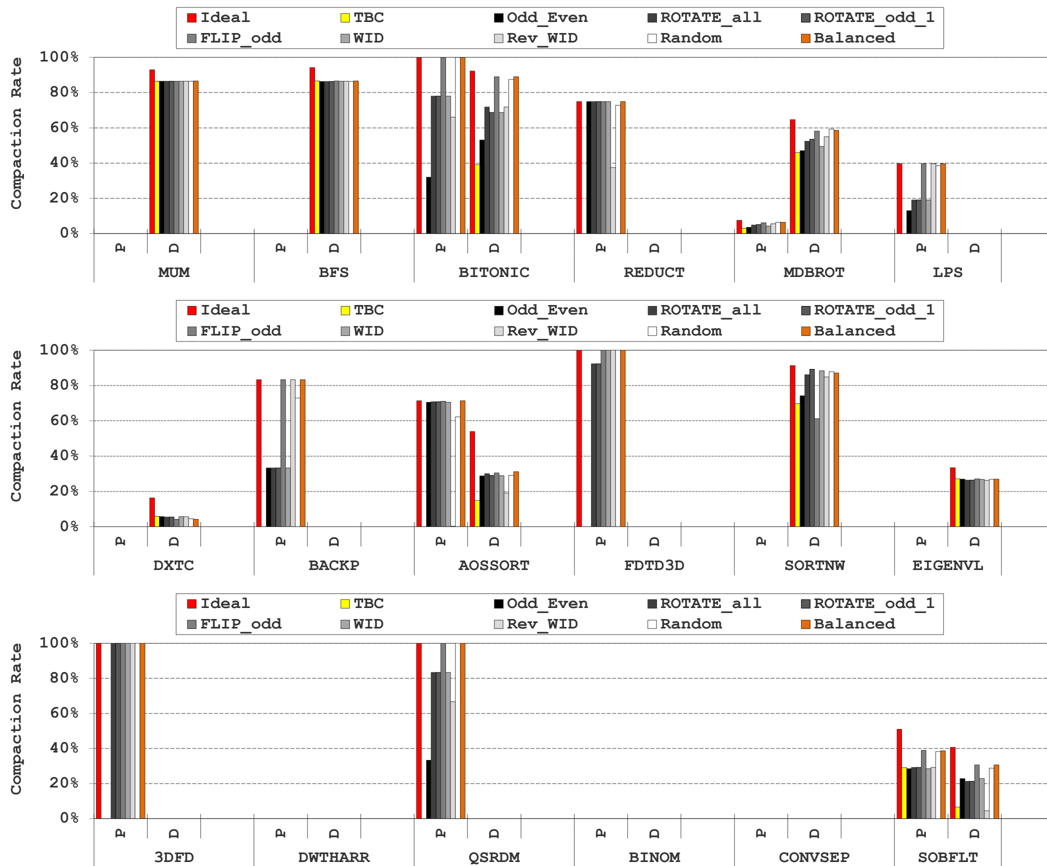


Figure 4.26: Compaction rate per branch type with different permutations.

forms TBC in terms of execution time, but skips some compaction opportunities by design.

4.4.3.1 Compactability

Figure 4.26 shows the compaction rate of P-branches and D-branches achieved with different permutations (including baseline TBC) across the 18 divergent benchmarks. Looking at P-branches first, 10 of the 15 benchmarks

contain compactable P-branches (see Section 4.3.3). Despite their heuristic nature, all but one of the permutations significantly improve P-branch compactability over the 3.2% compaction rate possible without SLP; SLP with *Odd_Even* cannot compact FDTD3D and 3DFD. However, only SLP with the carefully designed *Balanced* permutation comes close to ideal compaction for P-branches, averaging a compaction rate of 71.5% (98% of the ideal 72.7% average compaction rate). *Balanced* precisely matches *Ideal* in 7 of the 10 benchmarks. In FDTD and MDBROT, *Balanced* is within 1.1% of ideal. SOBFLT is the only benchmark in which *Balanced* failed to to achieve near-ideal compaction rate (38.7% compared to 50.9%). *Balanced* was still within 1% of the best permutation (*FLIP_odd*) even on this benchmark. In contrast, the previously proposed permutations, *Odd_Even* and *Rev_WID*, only achieve an average of 28.9% and 52.8% compaction rate respectively (compared to the ideal 72.7% compaction rate). *Odd_Even* only works for a very specific pattern and is not robust across the applications. Randomized permutations do not perform well when there is a large fraction of active threads, which is further discussed below.

Figure 4.27 shows a breakdown of the compactability of P-branches depending on the fraction of threads that are active in the CTA after the branch point. The randomized *Rev_WID* [44] permutation is more effective than the naive *Odd_Even*, but *Rev_WID* still misses significant opportunities for compaction when half or more of threads are active (e.g., in BITONIC, REDUCT, MDBROT, AOSSORT and QSRDM). In fact, *Rev_WID* even performs worse

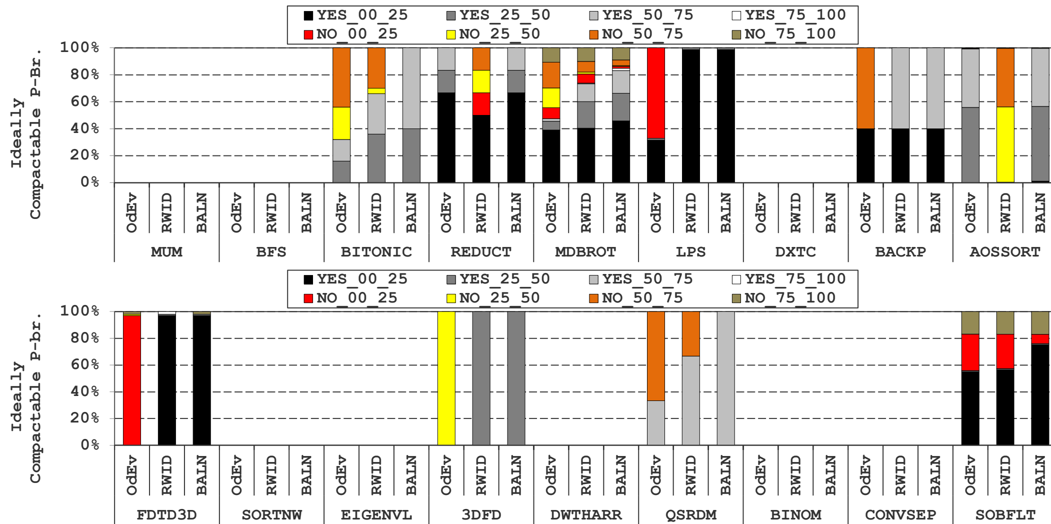


Figure 4.27: Breakdown of P-branch compaction rate by fraction of active threads across the CTA at the branch point. *(XX_YY_ZZ)* designates whether a branch is compacted or not (XX) for a particular fraction range of active threads in that path ($YY \leq$ ActiveThreads $< ZZ$). Note that the full 100% bar is equivalent to the *Ideal* bar in Figure 4.26. *OdEv*, *RWID*, *BALN* refers to *Odd_Even*, *Rev_WID*, and *Balanced* respectively.

than *Odd_Even* for REDUCT and AOSSORT. *Balanced* does well even in these challenging cases, missing few or zero opportunities for compaction (small to insignificant yellow, orange, and red portions in the figure).

Of the 18 benchmarks with divergent branches, 9 have divergent D-branches. As expected, baseline TBC does much better compacting D-branches than P-branches and has an average compaction rate of 42.5% compared to the ideal average of 64.4%. While the average compactability with the baseline TBC is reasonable overall, it is quite poor in 5 of the 9 benchmarks (BITONIC, MDBROT, AOSSORT, SORTNW, and SOBFLT), with an average of just

35.3%. SLP significantly improves compaction in these 5 benchmarks. *Balanced* provides the most improvement and shows its robustness by averaging 59.3% (86% of the ideal 68.5% average compaction in these 5 benchmarks). In fact, *Balanced* always achieve more than 97.9% of the best permutation in all benchmarks except DXTC. DXTC compacts best with no permutations because the active threads in DXTC are exhibited in groups of clusters, but the groups themselves are randomly scattered across the CTA which makes SLP less effective. As with P-branches, *Odd_Even* was the worst performer and even trailed the average of baseline TBC.

Figure 4.28 shows the compactability breakdown depending on fraction of active threads for D-branches. While the benefits are not as pronounced as with P-branches, *Balanced* again demonstrates that random permutations are a poor choice when a large fraction of threads are active. Note that with D-branches, SLP does slightly impair compactability in a few cases, but all are within 1% of baseline TBC. One possible optimization to prevent degrading baseline is to utilize compiler-support; SLP is disabled when the percentage of D-branches is above a threshold.

4.4.3.2 SIMD Lane Utilization

The definition of compaction rate ignores the magnitude of compaction and focuses solely on whether any reduction in the number of warps was achieved. In contrast, $SIMD_{util}$ ignores how often compaction was successful and instead represents the overall magnitude of compaction. As shown

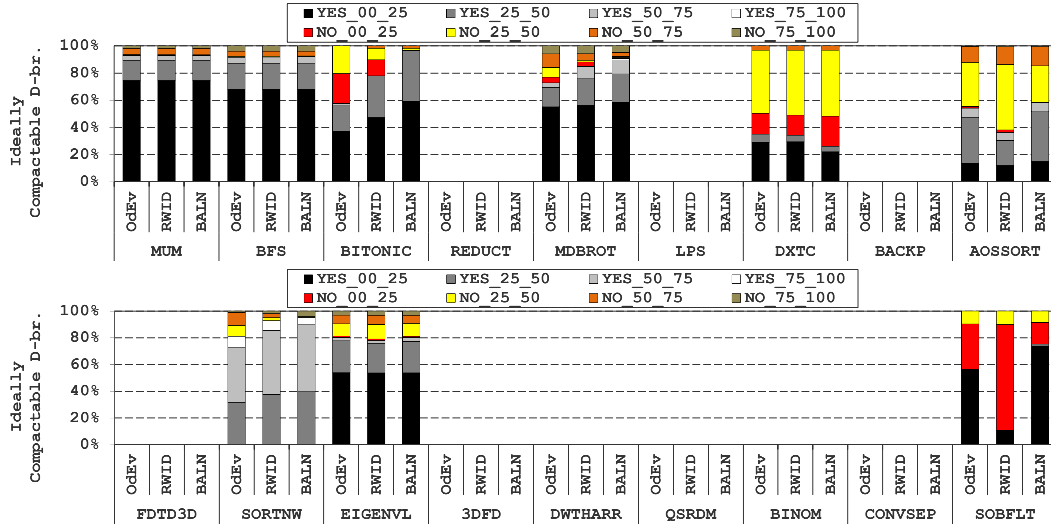


Figure 4.28: Breakdown of D-branch compaction rate by fraction of active threads in a path across the CTA.

in Figure 4.29, for instance, although there are multiple permutation methods that achieve the full compaction rate for BITONIC, REDUCT, FDTD3D, and 3DFD, the resulting $SIMD_{util}$ using these permutations is highly variable and only *Balanced* offers consistently high improvements.

Balanced most effectively reduces the number of warps overall, with average increases of 11.3% over the *No_TBC* baseline and 7.1% (max 34%) over TBC without SLP. *Balanced* is also either the permutation with the highest $SIMD_{util}$ (in 7 benchmarks) or within 99.4% of the best performing permutation in the other benchmarks. *Odd_Even* provides the smallest benefits, but still increases $SIMD_{util}$ by 6% and 2.1% (max 18%) compared to *No_TBC* and TBC respectively. *Rev_WID* [44] does better than *Odd_Even* with an aver-

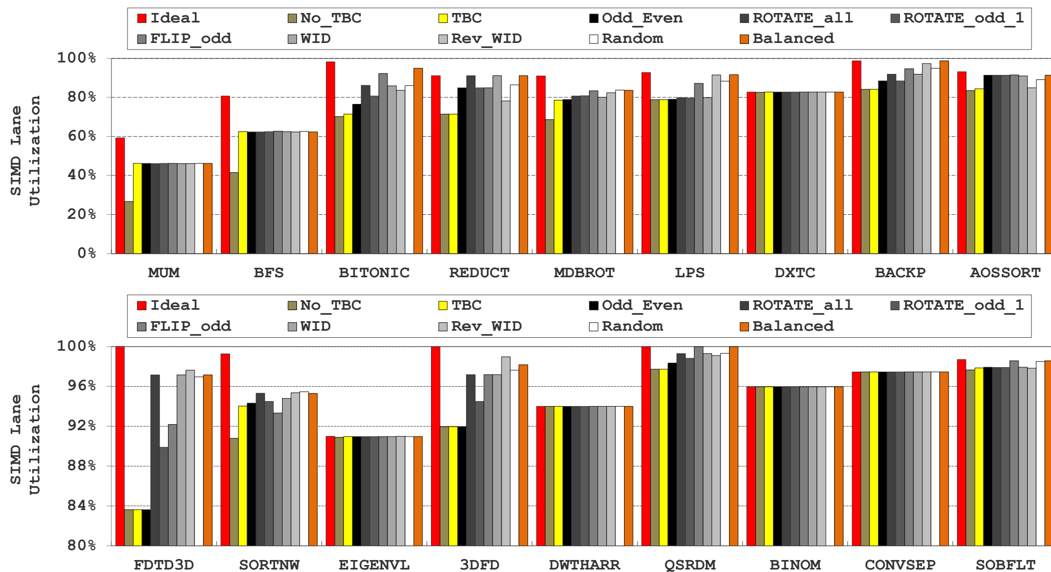


Figure 4.29: Average SIMD lane utilization with different permutations. Note that scale of the bottom chart above is $80 - 100\%$.

age 8.8% and 4.8% increase over *No_TBC* and TBC respectively. However, *Rev_WID* lacks robustness and significantly trails the best performing permutation by up to 14.3% (e.g., for BITONIC, REDUCT, AOSSORT, QSRDM). Among the 5 benchmarks that experienced a lowered compaction rate for their D-branches with SLP compared to baseline, all also experience a decrease in $SIMD_{util}$ for a some SLP permutations but of less than 0.1%.

4.4.3.3 Overall Performance

Figure 4.30 shows the performance of those 8 benchmarks that can be executed as-is in GPGPU-Sim. TBC is augmented with the 1b-Latest con-

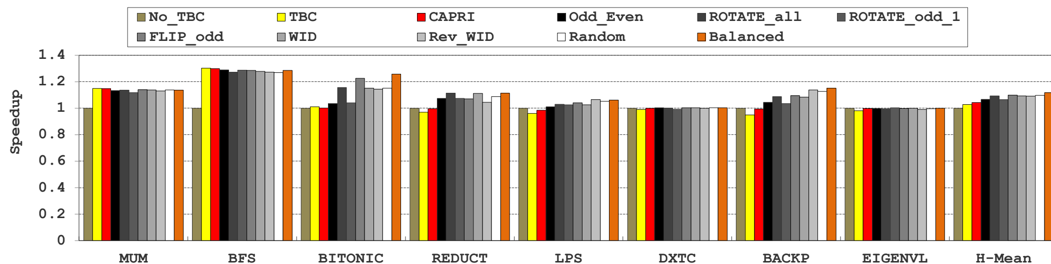


Figure 4.30: Speedup of compaction with different permutations over no-compaction.

figuration of CAPRI (Section 4.4) and the SLP mechanisms are implemented on top of CAPRI. *Balanced* provides the highest average IPC increase, outperforming baseline by 11.6% and CAPRI by 7% (both harmonic means). *Balanced* also exhibits the maximum speedup observed, improving BITONIC by 25.6% and BACKP by 15.2%. It is also the best performing SLP permutation on 4 of the 8 benchmarks and is always within 98.9% of the best permutation. In contrast, the second-best permutation, *FLIP_odd*, achieves an average speedup of 5% over CAPRI and in worst case achieves only 90.5% of the best performing SLP. While being effective for compacting BFS and MUM, TBC fails to utilize the P-branches for the other 6 benchmarks and performs worst among all compaction mechanisms.

Because SLP degrades the compaction rate and $SIMD_{util}$ of MUM, BFS, DXTC, and EIGENVL, performance for these benchmarks are decreased. *Balanced* and *FLIP_odd* showed the least degradation in performance (0.3% and 0.4% degradation compared to CAPRI without SLP, respectively). *Ro-*

tate_odd_1 caused the greatest average IPC degradation for these benchmarks (1.1%), but overall, even this permutation improved performance by 2.1% (harmonic mean). Although we do not evaluate the performance of the other 10 benchmarks, we expect the strong correlation between $SIMD_{util}$ and performance to apply to them as well.

4.4.3.4 Impact on the Memory System

Compaction involves dynamically rearranging threads from different warps and scheduling them together at the same time. While SLP itself does not disturb the memory coalescing capability (as discussed in Section 4.3.4), the dynamic formation of warps through compaction can degrade memory access behavior compared to a pipeline that does not compact at all. Among the evaluated benchmarks, those exhibiting a substantial increase in $SIMD_{util}$ and performance did show a noticeable increase in L1 misses, with an average increase of 6.9% and a maximum increase of 27% (BFS, as in Figure 4.24). Although L1 misses were more frequent than without compaction, the impact on L2 misses and memory traffic is negligible (1.3% average increase). These results are in line with the observations from prior work [23, 24, 25]. Because the benefits of compaction usually outweigh the increase in L1 traffic, overall performance is increased, as previously demonstrated.

4.4.3.5 Implementation and Energy-Efficiency of SLP

Enabling SLP on top of TBC requires storage for the permutation table and a small amount of logic for permuting the home SIMD lanes. The storage requirements for *Balanced* is just 5 bits for each of the 32 unique *XOR* masks, totaling 160 bits per GPU core.

The required logic is just a handful of *XOR* gates per lane. In addition, a single control bit is needed to allow a programmer to disable SLP for a kernel to maintain backward compatibility for codes that rely on the undocumented behavior of a single arbitration order between logical threads within a warp. Given that a 32-entry CAPT (1088-bits storage) consumes less than 10mW, the area/power overhead of SLP is expected to be trivial.

4.5 Discussion

This section summarizes some key discussion points that are related to compaction-based architectures. We start by presenting prior work that is closely related with the proposed schemes and discuss the impact of CAPRI and SLP on programmability, followed by possible future extensions of CAPRI and SLP.

4.5.1 Software-Based Permutation

Zhang et al. [65] proposed *reference redirection*, which is a software only technique that attempts to provide an alternative to hardware compaction. Reference redirection statically rearranges threads with similar control flow

into a common warp to mitigate the impact of control divergence. This software technique often incurs significant overhead for two main reasons. First, rearranging the threads has a runtime component that must be amortized over long-running threads and kernels. Second, the new thread groups can degrade memory coalescing behavior for the entire duration of the kernel. The key difference from hardware compaction is that reference redirection is static for the entire kernel, whereas compaction only dynamically rearranges threads in those basic blocks where it is effective. Thus, while reference redirection can degrade the memory performance of the entire kernel, hardware compaction executes identically to baseline SPE in basic blocks that do not diverge. A direct comparison between reference redirection [65] and SLP is beyond the scope of this dissertation.

4.5.2 Impact on Programmability

Despite its superiority in programmability and hardware efficiency, the SIMT model does require the programmer to possess deep understanding of the underlying GPU microarchitecture for maximum efficiency. Given that the actual granularity of thread execution is warps and not CTAs (which is the finest granularity of thread grouping exposed to the programmer, see Section 2.1), it is extremely difficult for a typical programmer to maximize SIMD efficiency (and thus overall throughput) unless he or she is fully aware of the *warp-based* execution model (e.g., the programmer should carefully write the program so that all of intra-warp threads follow similar control flow). By adopting

the *CTA-oriented* exection model of TBC, CAPRI, and SLP, the underlying microarchitecture will automatically squash out unused lanes and maximize SIMD lane utilization through compaction, as long as the programmer enforce intra-CTA threads to traverse through similar controls. This substantially improves programmability of a normal programmer because the notion of warps is less of a concern under the CTA-based model.

As mentioned in Section 4.3.4, however, some expert programmers of current GPUs occasionally rely on undocumented behavior for optimization and tuning. SLP may break applications that rely on the fact that in current GPU implementations the same logical thread location within a warp always wins arbitration when resource conflicts between threads in a warp occur. To preserve this arbitration behavior, which may be very desirable in some cases, (1) the programmer can explicitly disable SLP on a particular kernel, or (2) the logical ordering of the threads can be re-permuted to the original sequence prior to arbitration. The latter can be done trivially given the static and deterministic algorithm of SLP. Note that compaction itself already breaks this particular undocumented behavior.

4.5.3 Cost-Effective Implementation of CAPRI

While in Section 4.4.2.6 we concluded that the power overhead of CAPRI is negligible, an even simpler implementation of CAPRI is feasible. The key idea of such cost-effective CAPRI is to simply count the number of threads that are active when executing any branch instruction and only en-

force compaction when the number of active threads is smaller after branch divergence. In other words, all conditional but non-divergent branching points can naturally allow warps to bypass the compaction barrier (e.g., number of threads active before/after branching point will always be 32) and only those that are divergent will enforce synchronization. Such an implementation behaves the same as the 1-bit sticky adequacy configuration (Section 4.2.2) without the need for a fully-associative CAPT.

4.5.4 DPE with Compaction

Compaction-based GPU architectures can be augmented with the DPE model (Chapter 3) by naturally extending the CTA-wide reconvergence stack to accommodate both left/right paths of a divergent branch. Because compaction trades off SIMD utilization with thread-level parallelism (e.g., the more effective compaction works, the less number of warps to schedule), augmenting compaction with the DPE model may provide further benefits by enhancing path parallelism. However, all the applications we studied that exhibit divergence (Table 4.2) contain zero or very few interleavable branches and therefore DPE has no impact.

4.6 Summary

In this chapter I argue that previously proposed mechanisms to mitigate the negative impact of control divergence fall short because they introduce excessive synchronization, even when compaction provides no benefit. My dis-

sertation demonstrates that a large fraction of conditional branches cannot benefit from compaction because they happen not to diverge much or because their divergence pattern is not amenable to compaction. To overcome the fundamental deficiency of decreased performance from unnecessary synchronization, a dynamic hardware predictor that predicts whether a branch point is likely to adequately benefit from compaction has been proposed and evaluated. When the prediction is positive, all divergent warps that execute the branch will stall and wait for compaction. If the adequacy prediction is negative, compaction is bypassed and synchronization is avoided.

Prior compaction techniques provide benefit for highly-divergent applications but degrade performance in some cases (by up to 19%). The proposed CAPRI mechanism provides superior performance improvements when improvements are possible. At the same time, by correctly identifying lack of compaction-adequacy, CAPRI matches the performance of the baseline *No_TBC* to within $\pm 2\%$. With very small area overhead, CAPRI is able to improve performance by up to an average 7% (max 11%) on top of TBC on divergent workloads and avoid TBC's 10% average (max 19%) performance degradation in non-divergent cases.

In addition to the excessive synchronization overhead, the limited applicability of compaction is also a key concern. This chapter explains and quantitatively demonstrates that in many cases, the way threads are associated with SIMD lanes causes aligned divergence patterns that prevent compaction. The detailed analysis reveals that such alignment mainly originates

from non-data dependent, programmatic branches. Although diverging paths from these programmatic branches do not compact well as-is (an average of 3.2% compaction rate), there is substantial opportunity (72.7% compaction rate) if the fixed association of threads to lanes can be relaxed. Importantly, such programmatic branches are common in applications with irregular control (11 of the 18 benchmarks we evaluate).

I propose and evaluate SLP, which expands the applicability of compaction by reducing, and even eliminating, aligned divergence. SLP permutes the mapping of logical thread locations to physical SIMD lanes when a warp is launched. This breaks aligned divergence patterns resulting from conditionals that depend only on programmatic values. A novel and robust *Balanced* permutation technique has been proposed, which enables an average of 71.5% of programmatic branches to be compacted – 98% of the ideal 72.7%. As a result, SLP with *Balanced* achieves the highest $SIMD_{util}$ and performance of all compaction and permutation mechanisms across the 18 benchmarks we study.

Because the permutation scheme has minimal hardware overhead and does not directly impact any component other than determining the association of threads and lanes, I argue that it should become the default architecture for GPUs that utilize thread compaction.

Chapter 5

A Locality-Aware Memory Hierarchy

The previous chapters discussed the problem of SIMT control divergence and the mechanisms I developed to address these inefficiencies. In addition to control divergence, however, memory divergence due to irregular memory accesses is another key concern for throughput processors. This chapter demonstrates that memory divergence, coupled with the massive multithreading of throughput processors, results in highly inefficient caching and substantial waste in off-chip memory bandwidth utilization. I propose a locality-aware memory hierarchy [27] to remedy such inefficiency by adaptively adjusting the data fetching granularity from the off-chip memory, achieving better performance as well as better energy-efficiency.

5.1 Conventional Memory System Designs and Data Locality

This section reviews some key characteristics of the baseline coarse-grained (CG) memory system (Section 2.3) and its pros and cons in a throughput computing environment.

5.1.1 Coarse-Grained Memory Hierarchy

The *CG* memory system enables throughput processors to exploit programs with high spatial locality, increasing peak memory bandwidth and decreasing control overheads. Regularly structured, compute-intensive applications can readily utilize the high peak memory bandwidth and ample computational resources of GPUs to great effect. However, not all applications can be re-factored to exhibit regular control flow and memory access patterns. In fact, many emerging GPU applications suffer from inefficient utilization of off-chip bandwidth and compute resources [14, 66, 16]. Recent research has primarily focused on overcoming irregularity by improving SIMD resource utilization and latency tolerance [22, 67, 21, 23, 24, 25, 35, 20, 36, 26, 68, 37], but the memory bandwidth bottleneck still remains a significant issue in future throughput computing [69]. Despite the significance of achieving high utilization of expensive off-chip bandwidth, this issue has received little research attention. As a result, previous research assumes a memory system optimized for *CG* accesses, regardless of the architectural nature of throughput processors and application characteristics. My work provides an architectural optimization to throughput processors such that they can manage irregular memory access patterns more effectively.

5.1.2 Limitation of Coarse-Grained Memory Systems

Limited Per-Thread Cache Capacity. CPUs have traditionally employed a small number of threads that share a large on-chip cache hierarchy, which

Table 5.1: Per thread cache capacity of state-of-the-art CPUs and GPU.

Intel **Core i7-4960X** [70]	**IBM** **Power7** [71]	**Oracle** **UltraSparc T3** [72]	**NVIDIA** **Kepler GK110** [30]
32 KB L1 2 threads/core 16 KB/thread	32 KB L1 4 threads/core 8 KB/thread	8 KB L1 8 threads/core 1 KB/thread	48 KB L1 2,048 threads/core 24 B/thread

allows a significant fraction of the working set to be captured inside the cache. GPUs, on the other hand, utilize a very large number of concurrent threads to achieve high latency tolerance, but this limits the per thread cache capacity available on chip. This difference between CPUs and GPUs is summarized in Table 5.1, which illustrates the orders of magnitude smaller per thread cache space allocated within each GPU core. As detailed below, such limited on-chip capacity per thread leads to high cache contention and significantly constrains the average lifetime of cache blocks.

Temporal/Spatial Locality of GPU Applications. Figure 5.1 shows the distribution of *repeated accesses* across all cache blocks in the baseline memory system. Twelve of the 20 benchmarks suffer from poor cache block reuse because of low temporal locality and high on-chip storage contention. As a result, more than 50% of the L2 cache blocks are never reused before eviction.

Note that GPU cores do not always request data in full cache block granularity (e.g., the baseline cache block size of 128 bytes). For highly irregular memory accesses, it is common that the requested data size is smaller than

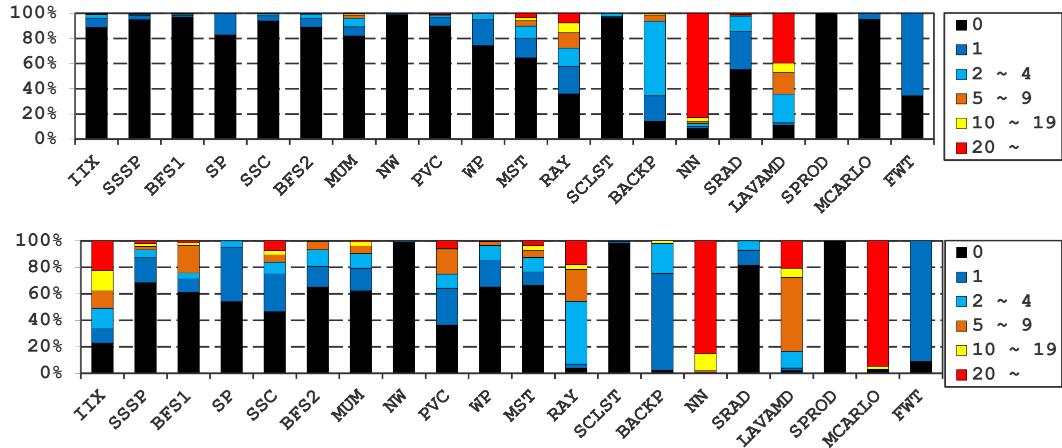


Figure 5.1: The distribution of repeated accesses to cache blocks in the L1 (top) and L2 (bottom) caches (using a CG-only memory hierarchy). Black regions represent *zero* reused cache blocks.

the cache block width; a request can be as small as 32 bytes in CUDA (see Section 2.3). Because of the low temporal locality, cache blocks that are filled in response to partial-block requests frequently exhibit poor spatial locality as well; the data that was filled but not explicitly requested is never used. Figure 5.2 shows the spatial utilization of L1/L2 cache blocks throughout their lifetime; each cache block is logically divided into four distinct regions (or *sectors*) with each region containing 32 bytes of data (consecutive byte addresses). We count how many distinct regions have actually been referenced before a block gets evicted. While some applications (e.g., SPROD, MCARLO, FWT, etc) utilize most of the fetched data, thereby maximizing the benefits of *CG* memory accesses, others (e.g., IIX, SSSP, etc ...) over-fetch memory sectors, inefficiently utilizing the *CG*-only memory hierarchy.

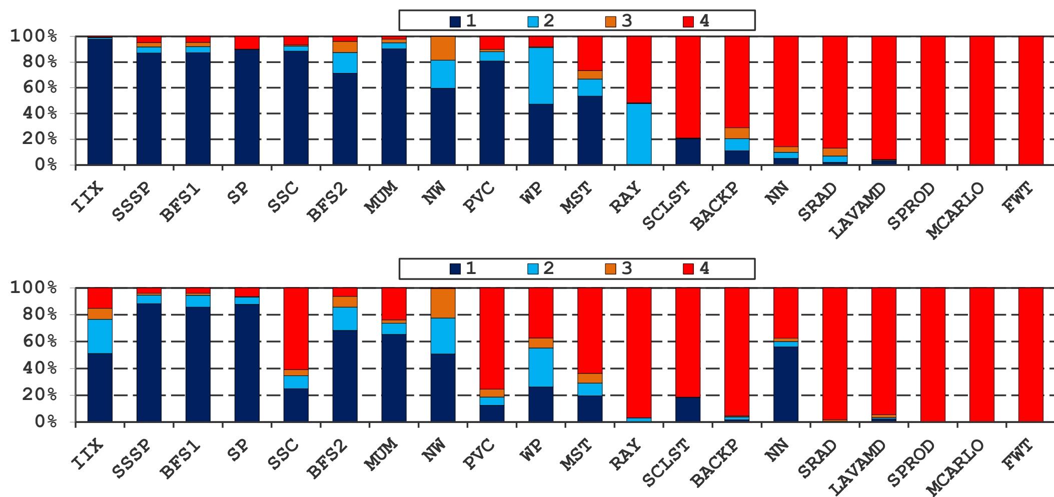


Figure 5.2: The number of sectors referenced in L1 (top) and L2 (bottom) cache blocks using a *CG*-only memory hierarchy. Each 128 bytes cache block is logically divided into four 32 bytes sectors (equivalent to the *smallest* data request generated by GPU cores). Dark blue regions, representing those that have only a *single* sector referenced, exhibit only 25% of spatial utilization whereas red regions represent 100% utilization.

Waste in Off-Chip Memory Bandwidth Utilization. In general, regularly structured programs with high spatial and temporal locality use most or all of the sectors within each cache block, effectively utilizing the *CG* memory accesses of the baseline GPU memory hierarchy. The massively multithreaded nature of GPUs, however, allows little cache capacity per thread, resulting in high cache miss rates and reducing the amount of temporal locality that can be exploited for certain applications. Such behavior, combined with the *CG*-only memory hierarchy, significantly over-fetches off-chip data for irregular applications, wasting memory bandwidth, on-chip storage, and DRAM power. The goal of this work is to minimize the amount of useless data fetching from the

off-chip DRAM, thereby maximizing the bandwidth utilization and improving both performance and energy-efficiency.

5.2 Designing a Locality-Aware Memory System

This section presents the first memory hierarchy design that can efficiently handle fine-grained irregular memory access patterns and scatter-gather programs in modern GPU architectures. I propose a reactive and efficient memory system that is *locality-aware*, such that it can cater to the behavior of irregular GPU programs. Prior work has used the dynamic estimation of *spatial* data locality for selective fine-grained (FG) memory accesses (e.g., fetching data smaller than cache block granularity) in control-intensive, general-purpose chip multi-processor environments [73, 74]. While these techniques are successfully deployed for CPUs, they fall short for massively multithreaded throughput-oriented GPUs because many emerging GPU applications with irregular control/memory accesses exhibit very low temporal locality and caching efficiency. I propose the *locality-aware memory hierarchy* (LAMAR) to provide the ability to tune the memory access granularity for GPUs with small implementation overhead. This section first discusses the possibility of statically making CG/FG decisions (guided by a profiler/autotuner) to best match the memory access granularity with application characteristics. A scalable, low-cost hardware predictor is then presented, which adaptively adjusts the memory access granularity without programmer or runtime system intervention.

In general, LAMAR maintains the advantages of CG accesses for programs with high spatial and temporal locality, while selective FG accesses reduce over-fetching and enable more efficient off-chip bandwidth utilization. By using multiple granularity memory accesses in a manner appropriate for the GPU memory hierarchy, LAMAR improves the efficiency of a wide range of GPU applications, significantly improving memory bandwidth, energy-efficiency, and overall performance. Also, the implementation of LAMAR is kept transparent to the user and without major changes to the underlying microarchitecture, easing its adoption into future GPU systems.

5.2.1 Fine-Grained Data Management

Coarse-grained accesses can be useful, as they reduce miss rates and amortize control costs for spatially and temporally local requests. In the absence of high locality, FG memory accesses avoid unnecessary data transfers, save power, and improve system performance. This section introduces some key microarchitectural structures that enable the FG management and storage of data.

Sub-ranked Memory. A conventional GPU memory system uses multiple DRAM chips organized in a rank to provide coarse-grained accesses to memory (Section 2.3, Figure 2.4). In order to exploit the benefits of FG accesses for irregular workloads, LAMAR must reduce the minimum access granularity to off-chip memory. To do so, LAMAR leverages a *sub-ranked* memory system

to non-intrusively allow fine-grained memory requests. The sub-ranked memory system adopted by LAMAR is inspired by many prior works, including HP's MC-DIMM (multi-core dual in-line memory module) [75, 76], Rambus's micro-threading [77] and threaded memory module [78], the mini-rank memory system [79], and Convey's S/G DIMM (scatter/gather dual in-line memory module) [80]. In a sub-ranked DIMM, peripheral circuitry is used to divert memory command signals to a sub-rank of DRAM chips without changing the DRAM structure itself. Figure 5.3 shows the sub-ranked memory system used for LAMAR, which provides a minimum access granularity of 32 bytes, equivalent to the smallest data request generated by a GPU core [28]. Because LAMAR provides an adaptive access granularity to suit program needs, *CG* memory requests are also allowed. Note that each *CG* access in LAMAR requires the memory scheduler to issue multiple 32 bytes data requests, which places more pressure on the address/command bus than the baseline configuration without sub-ranking [73]. The command signaling bandwidth is therefore doubled in order to fully utilize the off-chip data bandwidth. Increasing command bandwidth is a matter of system optimization and is already being deployed in commercial high-end systems such as Black Widow [81], FB-DIMM [82], S/G DIMM [80], and others.

Fine-Grained Cache Architecture. Fine-grained memory accesses require some cache changes to maintain *FG* information in the on-chip memory hierarchy. LAMAR utilizes a simple sector cache [83] to enable the on-chip

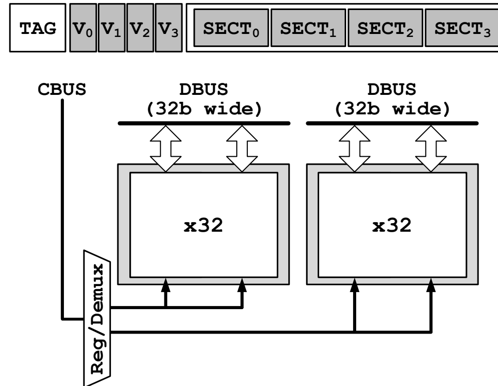


Figure 5.3: A memory system with two sub-ranks, providing a 32 byte (32b $\times$ 8-bursts) minimum access granularity. When used with a sector cache, the size of each sector is 32 bytes.

management and storage of FG data. The sector cache partitions each cache block into sectors, each with its own validity meta-data; this allows for data to be managed at a granularity finer than a cache block. Figure 5.3 illustrates how data are partitioned and stored into a sectored cache block. The sector cache used for LAMAR partitions each cache block into four 32 bytes sectors.

5.2.2 High-Level Overview of LAMAR

LAMAR uses a sector cache (both L1 and L2) and a sub-ranked memory system[1] in order to demonstrate the full benefits of our proposed scheme (Figure 5.4). The width of each sector, as well as the minimum access granularity of the sub-ranked memory system, is equivalent to the smallest data

[1]We also explore the implications of LAMAR with minimum changes to the GPU architecture by only using a sectored L1/L2 cache *without* a sub-ranked memory system, the result of which is detailed in Section 5.3.2.5.

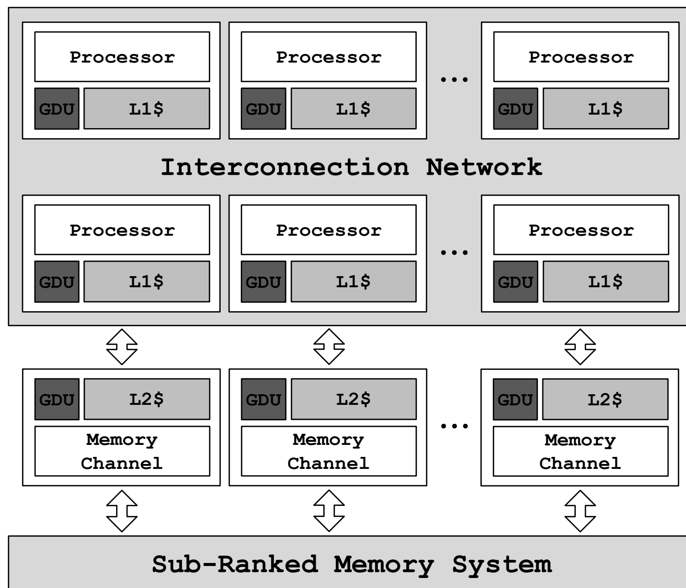


Figure 5.4: The proposed LAMAR GPU architecture. Each on-chip cache is sectored and is augmented with a GDU; off-chip memory is sub-ranked in order to allow fine-grained accesses.

request size generated by the address coalescing unit within the GPU core, which in current generation of GPUs is 32 bytes[2]. Each cache is augmented with a *granularity decision unit* (GDU) that determines the access granularity of each cache miss. In the baseline *CG*-only memory system, all cache misses are requested at a cache block granularity, whereas LAMAR uses the GDU to determine which access granularity best suits the application.

[2]Enabling a minimum access granularity smaller than 32 bytes requires restructuring the address coalescer in the GPU core. In this dissertation, I leverage the current GPU core architecture as-is to demonstrate the benefits of LAMAR while minimizing the changes to the current GPU architecture.

Statically-Driven GDU. LAMAR provides the programmer the option to tune the access granularity by statically designating whether to fetch all program data at a coarse or fine granularity. This decision may be guided by profilers/autotuners and is sent to the runtime system to update each GDU (e.g., through compiler options, APIs, etc.). Skilled programmers can therefore configure the GDU as appropriate to the application's needs, achieving optimal bandwidth utilization and energy-efficiency. As detailed in Section 5.3.2, we find the *average number of sectors referenced within a cache block* (Table 5.4, Avg_{sec}) to be a good metric for characterizing a program's access granularity.

Dynamically Adaptive GDU. Despite the advantages of a statically-driven GDU, identifying and specifying the optimal access granularity requires both extra effort from the programmer and system support. To this end, I propose a hardware-only mechanism that dynamically derives the optimal access granularity at runtime, achieving comparable benefits of the statically-driven GDU in a robust manner across all studied applications.

Previous Work. Previous work exploits adaptive granularity memory accesses in a multi-core processor [74] using a spatial-pattern predictor [84, 85] (SPP) in place of the GDU. Spatial pattern prediction uses a pattern history table (PHT) to collect and predict likely-to-be-used sectors upon a cache miss. Each cache block in an SPP-based system is augmented with a set of *used bits* that designate whether a given sector has been referenced or not. When a

cache block is evicted, the corresponding used bit information is committed to the PHT. Future misses to each block query the PHT to determine which sectors are likely to be referenced in the future, allowing targeted sector fetches. Details of the microarchitectural aspects of the SPP can be found in [74, 85].

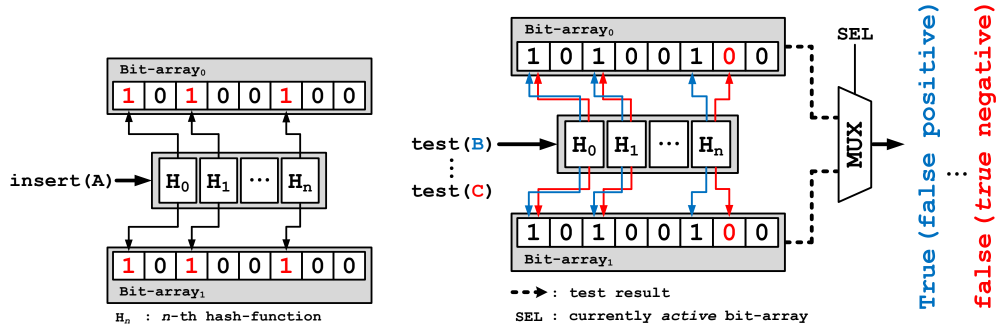
While spatial pattern prediction has been effectively employed in multicores, Section 5.3 quantitatively demonstrates why the SPP is not cost-effective in a massively multithreaded GPU environment. As pointed out in previous literature [35, 86], many GPU applications do not cache well and suffer from high cache miss rates and low block reuse. Such low caching efficiency occurs both due to streaming data accesses and also because the threads contend for cache resources and constrain the effective on-chip storage available to each thread. While massive multithreading enables GPUs to be highly latency tolerant, it comes at the cost of poor cache performance, which, combined with the CG-only memory system, wastes memory bandwidth and can limit system energy-efficiency.

The SPP is not as effective for multi-granularity access in GPUs as it is in CMPs because the high cache turnover rate and low cache block reuse of GPUs significantly lowers the ability of the SPP to learn dynamic behavior. While the SPP accurately estimates the spatial locality of data, it is not robust in the presence of low temporal locality (Figure 5.1) and poor cache performance as discussed in Section 5.3.2.1.

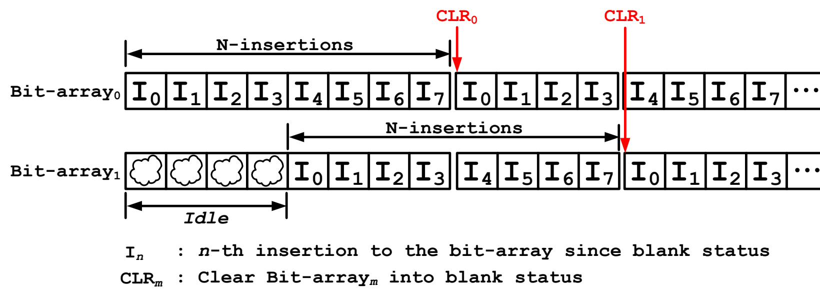
5.2.3 Bi-modal Granularity Predictor

In general, I observe that the SPP fails to provide robust prediction quality and high energy-efficiency to a wide range of GPU applications. While a more sophisticated prediction algorithm could potentially enhance the effectiveness of SPP, the complexity and high area overhead of a modified SPP will not scale to a many-core environment. I propose a simple, lightweight *bi-modal granularity predictor* (*BGP*) that is much more suitable for throughput-oriented architectures.

Key Idea and Observation. The main inefficiency of spatial pattern prediction is that the spatial locality information tracked by the PHT is useless for cache blocks with low temporal locality. With this in mind, the proposed *BGP* microarchitecture is structured such that it estimates *both* the temporal and spatial locality of missed cache blocks and determines whether to fetch *all* of the sectors within the cache block (*CG*-mode) or *only* the sectors that are requested (*FG*-mode). The key observation behind *BGP* is that for cache blocks with poor temporal locality, it is sufficient to fetch only the sectors that are actually requested (on-demand) because the other sectors will most likely not be referenced during their lifetime (e.g., cache blocks with *zero* reuse in Figure 5.1). Meanwhile, blocks with both high temporal and spatial locality make effective use of coarse-grained accesses, such that a simple bi-modal prediction is sufficient to greatly improve memory system performance and efficiency.



(a) *Inserting* an element. (b) *Testing* whether an element is within the set.



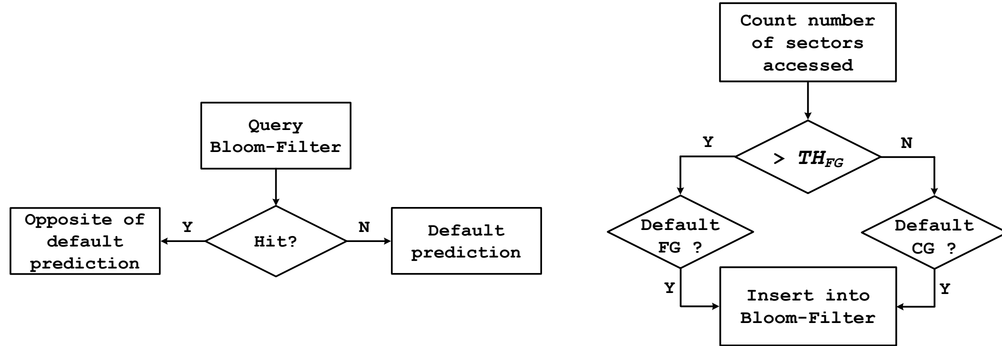
(c) Operation of a dual bit-array bloom filter.

Figure 5.5: A microarchitectural overview of the proposed dual bit-array bloom filter. Note that A, B and C in (a, b) are distinct values. All insertions into the bloom filter are applied to *both* bit-arrays (a), except during the initial idle period of *Bit-array*$_1$ (c). To minimize the false positive rate, each bit-array is *cleared* after N insertions at which point the *active* bit-array (determined by SEL in (b)) is swapped. This dual bit-array microarchitecture allows the bloom filter to retain at least a $\left(\frac{N}{2}\right)$ insertion history from the newly active bit-array, avoiding periods where it contains zero history.

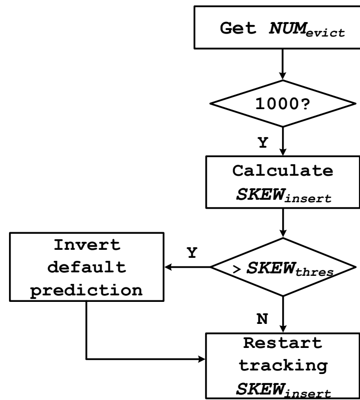
Microarchitecture. The lightweight *BGP* microarchitecture dynamically estimates if each missed cache block has enough locality to warrant a coarse-grained fetch. The storage of *BGP* is implemented using a *bloom filter* [87] to minimize the cost of tracking the multitude of cache blocks in the system. A

bloom filter is a space-efficient probabilistic data structure that is used to *test* whether an element is a member of a *set*. It consists of a bit-array with m-bits and n-hash functions. An element is *inserted* (Figure 5.5(a)) into the set by calculating n different hashes of the element and setting the corresponding bits of the bit-array. Testing if an element belongs to the set is done by feeding it to the n hash functions and checking if all the corresponding bit positions are 1s (Figure 5.5(b)). If any of the queried bits is 0, the element is definitely not a member of the set (true negative) while all 1s indicates either that the element actually was inserted to the set (true positive) or that there are collisions with other elements of the set (false positive). The false positive rate of a bloom filter is determined by the number and type of hash functions chosen and the size of the bit-array. For the purpose of the BGP, a bloom filter is used to track the set of evicted blocks having low (or high) locality, using their respective block address as the inserted element.

In order to temporally degrade old insertions and maintain a certain amount of locality history, the locality predictor is implemented using a *dual bit-array* microarchitecture as detailed in Figure 5.5. This dual bit-array bloom filter uses two temporally overlapped bit-arrays that are periodically cleared and swapped in order to eliminate interference due to stale locality data. This structure has several implementation advantages. First, the dual bit-array bloom filter allows for the removal of aging elements in an application appropriate manner without resorting to more expensive bloom filter variants (such as a counting filter [88]). Also, the rolling history of the dual bit-array



(a) The CG/FG prediction mechanism. (b) The BGP insertion mechanism.



(c) Default prediction inversion.

Figure 5.6: The prediction algorithm of the BGP. $SKEW_{insert}$ is evaluated every thousand cache block evictions (NUM_{evict}).

naturally captures temporal locality information. Finally, because the dual bit-array structure periodically resets, it allows us to tailor the default insertion/prediction mode (CG or FG) to dynamic phase behavior in order to reduce the false positive rate, as described below.

Prediction Mechanism. Figure 5.6 summarizes how the bi-modal granularity predictor operates and when and how evicted blocks are inserted into the bloom filter. The *BGP* contains a *default* prediction (*CG* or *FG*) that determines what kind of evicted blocks are inserted into the filter (and the corresponding prediction upon a query to the filter). The *CG*/*FG* fetch decision is made by querying the bloom filter with the missed block's address—upon a miss, the querying cache block has the opposite locality characteristics to the blocks inserted into the bloom filter, so *BGP* grants the default prediction. For those queries that hit in the bloom filter, the *BGP* predicts the opposite of the default prediction (Figure 5.6(a)).

The *BGP* uses the number of sectors accessed as means to approximate the locality of a cache block, rather than the number of accesses to the cache blocks, for simplicity in design. When a cache block is evicted, the associated sector used-bit information is examined to estimate the block's locality—if the number of sectors accessed is below a pre-determined threshold (TH_{FG}), that block is estimated as having low locality and high locality otherwise (Figure 5.6(b)). This locality estimate is compared with the *BGP*'s current default prediction in order to determine whether the eviction should be inserted into the filter.

When the percentage of evicted cache blocks inserted into the bloom filter ($SKEW_{insert}$) is high, each filter will be filled quickly and the *BGP* will track little temporal history (Figure 5.5(c)). Inspired by the intuition of *agree predictors* [89], the *BGP* rotates the default prediction whenever $SKEW_{insert}$

Table 5.2: Configuration parameters of *BGP* microarchitecture.

Bit-array size	2K bit per bit-array (4K bit per *BGP*)
Refresh period	Every 512 insertions
# of hash functions	6
Hash function	Byte-sliced XOR [90]
$\boldsymbol{TH_{FG}}$	2 sectors
$\boldsymbol{SKEW_{thres}}$	0.7

is higher than a pre-determined threshold ($SKEW_{thres}$) in order to avoid the bloom filter from being rapidly saturated by an overwhelming number of insertions (Figure 5.6(c)). Table 5.2 summarizes the microarchitectural parameters used for the baseline *BGP* configuration. The *BGP* bloom filter hash functions are inexpensively implemented in hardware by byte-slicing each evicted address and *XOR*-ing the slices together [90]. Overall, the prediction accuracy of *BGP* is shown to be relatively insensitive to these parameters, unless the bit-array size is less than 2K bits or the refresh period is less than a quarter of the bit-array size.

5.2.4 Summary of the Benefits of BGP

The benefits of the proposed *BGP* are twofold. First, by granting *FG* accesses for only those accesses that have past history of low temporal locality, applications with good caching behavior (e.g., most of the sectors are utilized) or those with a working set fitting well in the cache (e.g., low miss rates and thus low evictions within a timeframe) are guaranteed to fetch data in *CG*-mode, maintaining the benefits of the *CG*-only memory system. Second, the bloom filter based *BGP* provides a cost-effective mechanism to determine the

Table 5.3: Simulator configuration for LAMAR evaluation.

Number of GPU cores	15
Threads per GPU core	1536
Threads per warp	32
SIMD lane width	32
Registers per GPU core	32768
Shared memory per GPU core	48KB
Warp scheduling policy	Oldest warp first [23]
L1 cache (size/associativity/block size)	16KB/4-way/128B
L2 cache (size/associativity/block size)	768KB/16-way/128B
Memory bandwidth	179.2 GB/s
Memory controller	Out-of-order (FR-FCFS)

access granularity, as opposed to SPP-based schemes that require a separate PHT and complex control/update logic.

5.3 Evaluation

This section presents the evaluation methodology, followed by a detailed evaluation of LAMAR.

5.3.1 Methodology

Simulation Model. LAMAR is modeled using GPGPU-Sim [47, 46], a cycle-level performance simulator of a general purpose GPU architecture that supports CUDA 3.1 and its PTX ISA. The memory hierarchy of GPGPU-Sim is augmented with sectored L1/L2 caches and DrSim [91, 92], a detailed DRAM simulator that supports sub-ranked memory systems.

The DRAM model is configured to adhere to the GDDR5 specification [40], except for the bank-grouping effects (which are projected to be elim-

inated in future GDDR products [40, 93]). To demonstrate how the *BGP* is affected by limited hardware resources (e.g., dual 2K bit-arrays), the SPP and *BGP* with unrealistically large histories (1M-entries) are also simulated; these impractical designs are denoted by SPP and BGP_{inf} henceforth.

In general, the GPU simulator is configured to be similar to NVIDIA's GTX480 [38] using the configuration file provided with GPGPU-Sim [48]. Key microarchitectural parameters of the baseline configuration are summarized in Table 5.3; we explicitly mention when deviating from these parameters for the sensitivity studies in Section 5.3.2.5.

DRAM Power Model. This work uses the detailed power model developed by Micron [94] and the DRAM physical parameters have been referenced from a Hynix GDDR5 specification [40]. Our power model is summarized in Equation 5.1, and includes the background power, refresh power (P_{REF}), activation & precharge power (P_{ACT_PRE}), read power (P_{RD}) and write power (P_{WR}). The background power includes precharge standby power (P_{PRE_STBY}) and active standby power (P_{ACT_STBY}). Read and write power includes the power consumed by the DRAM bank (P_{RD_BANK}) and by the IO pins (P_{RD_IO}).

$$\begin{aligned} P_{GDDR5} &= \underbrace{P_{PRE_STBY} + P_{ACT_STBY}}_{Background\ Power} + P_{REF} + P_{ACT_PRE} \\ &\quad + \underbrace{P_{RD_BANK} + P_{RD_IO}}_{P_{RD}} + \underbrace{P_{WR_BANK} + P_{WR_IO}}_{P_{WR}} \end{aligned} \tag{5.1}$$

GPU Processor Power Modeling. This study is concerned primarily with the performance and efficiency of the memory hierarchy. To evaluate how LAMAR affects the overall system energy-efficiency, however, we model the GPU processor power using the analytical IPC-based power model suggested by Ahn et al. [75]. The peak power consumption of each GPU core is extracted using GPUWattch [53]. The leakage power of the system (including GPU processors and DRAM) is estimated to be 59W. The peak dynamic power consumption per GPU core is estimated to be 9.5W, out of which 2.3W are constant power that does not scale with IPC. Such simple IPC-based power modeling offers $> 90\%$ agreement with GPUWattch and is used to estimate the overall system efficiency in Section 5.3.2.4.

Benchmarks. LAMAR is evaluated with 32 benchmarks from Rodinia [14], CUDA-SDK [3], MapReduce [66], LonestarGPU [16], and the benchmarks provided with GPGPU-Sim [47]. This chapter focuses on the 20 applications that exhibit noticeable differences across different schemes for brevity (Table 5.4). All benchmarks are simulated to completion, with the exception of SSSP, SP, PVC, SCLST, and FWT; due to the long simulation time of these applications, we execute them only up to the point where IPC is saturated with small variation among different iterations of the kernel. The 20 chosen benchmarks are categorized as either being *FG-leaning* or *CG-leaning* based on the average number of sectors accessed within all L1/L2 cache blocks (Figure 5.2) —

Table 5.4: Benchmarks studied for LAMAR evaluation. Avg_{sec} refers to the average number of sectors accessed across all cache blocks.

Abbreviation	**Description**	**#Instr.**	$\boldsymbol{Avg_{sec}}$	**Ref.**
IIX	Inverted index	1.8B	1.09	[66]
SSSP	Shortest paths	1.5B	1.24	[16]
BFS1	Breadth first search	3B	1.25	[16]
SP	Survey propagation	1.3B	1.28	[16]
SSC	Similarity score	4.9B	1.46	[66]
BFS2	Breadth first search	469M	1.48	[14]
MUM	MUMmerGPU	149M	1.49	[63]
NW	Needleman-Wunsch	220M	1.67	[14]
PVC	Page view count	5.4B	1.75	[66]
WP	Weather prediction	365M	2.00	[47]
MST	Min. spanning tree	5B	2.39	[16]
RAY	Ray-tracing	750M	3.29	[47]
SCLST	Streamcluster	4.1B	3.41	[14]
BACKP	Back propagation	196M	3.62	[14]
NN	Neural network	78M	3.65	[47]
SRAD	Structured grid	8.5B	3.88	[14]
LAVAMD	N-body	22B	3.89	[14]
SPROD	Scalar-product	25M	3.99	[3]
MCARLO	Monte-carlo	1B	3.99	[3]
FWT	Fast-walsh-transform	3.9B	4.00	[3]

applications that average more than two sectors accessed per cache block are categorized as CG-leaning (and FG-leaning otherwise).

5.3.2 LAMAR Results and Analysis

This section evaluates LAMAR, considering its impact on off-chip traffic, cache efficiency, the improvements that LAMAR brings about in overall performance and energy-efficiency, and its implementation overhead. We also discuss the variation of off-chip traffic to key microarchitectural parameters as a sensitivity study. Five different GPU memory hierarchy designs are an-

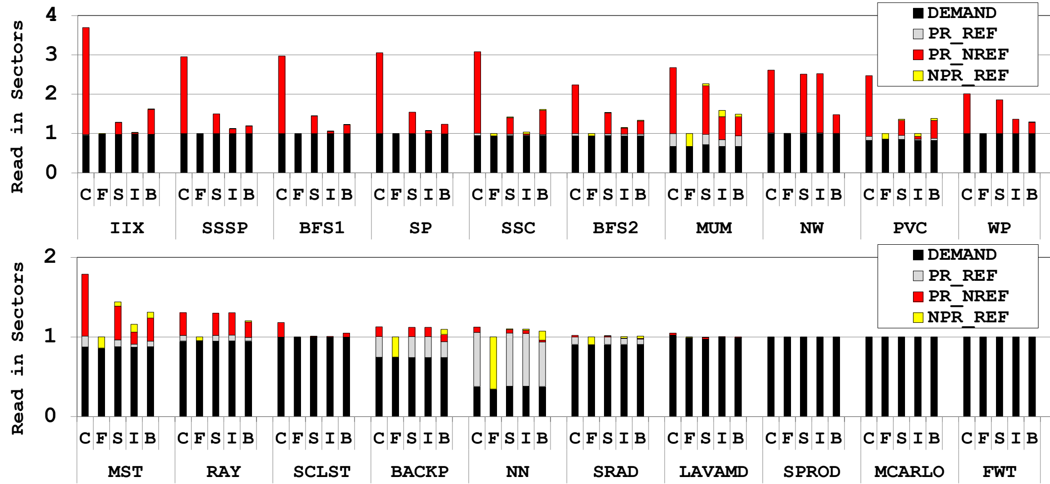


Figure 5.7: The number of sectors read into the L1 caches, categorized based on prediction quality (normalized to the FG-only scheme).

alyzed: *CG*-only, *FG*-only, SPP, BGP_{inf} and *BGP*, which are denoted by C/F/S/I/B, respectively, in all figures throughout this section. A LAMAR configuration based on static GDU decisions is equivalent to the best of *CG*-only and *FG*-only for each application. Note that the same dataset for both the profiling and measurement run are used. All average values are based on harmonic means.

5.3.2.1 Prediction Quality, Traffic, and Caching

The GDU of LAMAR determines whether a cache block should be fetched in *CG* or in *FG* mode. It can therefore predict to: 1) correctly fetch sectors that are actually referenced (*PRED_REF*), 2) incorrectly fetch sectors that are not referenced (*PRED_NREF*), and 3) incorrectly not fetch sectors that are referenced later (*NPRED_REF*). We therefore categorize each fetched

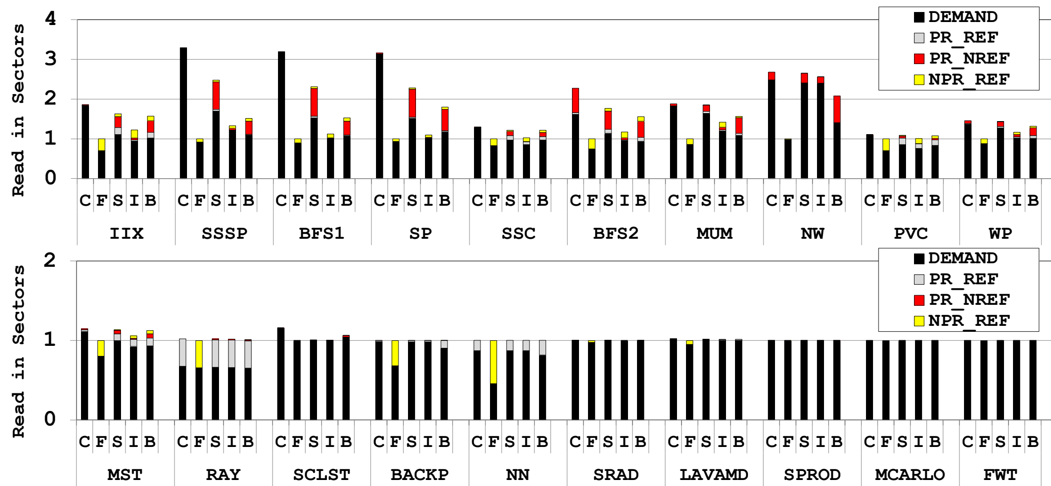


Figure 5.8: The number of sectors read into the L2 caches, categorized based on prediction quality (normalized to the FG-only scheme).

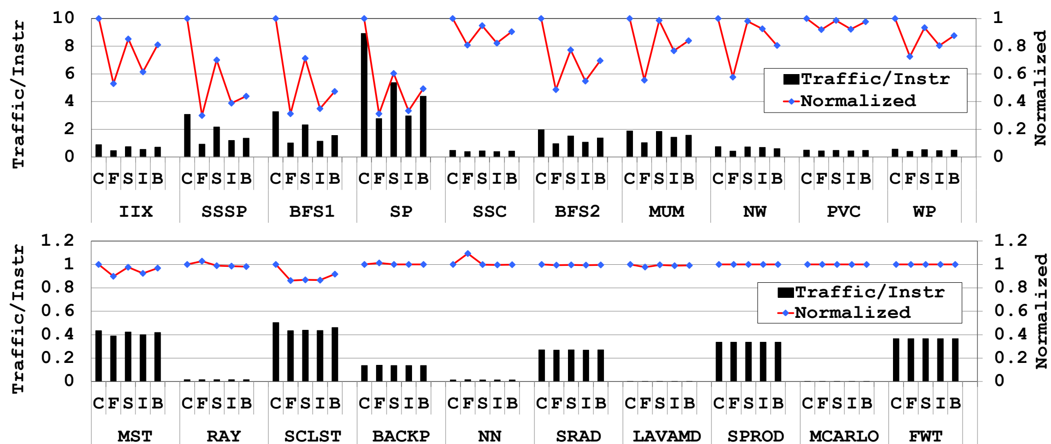


Figure 5.9: Byte traffic to DRAM (both read/write) normalized by 1) the number of instructions (left axis) and 2) by the traffic/instr. of the CG-only scheme (right axis).

sector as either being fetched on-demand from the upper level (*DEMAND*) or

based on prediction (Figure 5.7 and Figure 5.8)[3]. The overall read/write traffic and the associated cache miss rates are depicted in Figure 5.9, Figure 5.10, and Figure 5.11. Overall, the *FG*-only scheme has the smallest off-chip traffic thanks to its conservative fetch decision. This reduced traffic, however, comes at the cost of a significant portion of sectors being *NPRED_REF* with increased miss rates for some applications. NN, for instance, has 65%/54% of its L1/L2 sectors being fetched *NPRED_REF*. Because these sectors would have been pre-fetched had the initial access been predicted as *CG*-fetches, memory access behavior and caching efficiency are degraded, potentially leading to performance penalties for certain applications (see Section 5.3.2.2 for details). *CG*-leaning applications generally contain less overfetched data (even with the *CG*-only scheme), with only 13%/3% more sectors fetched to L1/L2 compared to the *FG*-only scheme. For *FG*-leaning benchmarks, however, *CG*-only falls short by having 171% and 93% more L1/L2 read-in traffic than *FG*-only, most of which is due to the large number of mispredicted *PRED_NREF* sectors.

Unlike *CG*-/*FG*-only schemes, dynamically-driven LAMAR is able to balance the benefits of both *CG* and *FG* accesses. All three LAMAR predictors reduce off-chip traffic significantly without degrading the memory access behavior of *CG*-leaning applications. SPP is the least effective mechanism among the three, having 60%/72% more L1/L2 read-in sectors than *FG*-only,

[3]Note that sectors requested from the L1 to L2 cache are interpreted as *DEMAND* sectors from L2's perspective, even though these sectors can be *PRED_REF*, *PRED_NREF*, and *NPRED_REF* from the L1's point of view.

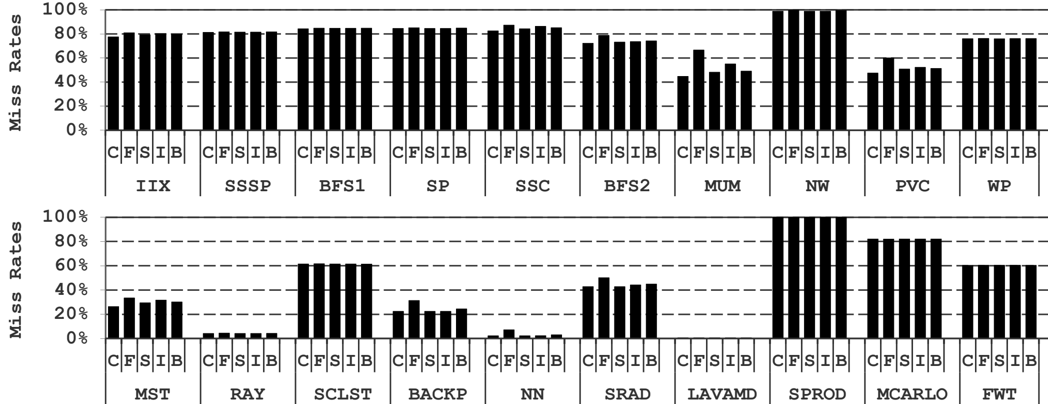


Figure 5.10: Changes in L1 cache miss rates with LAMAR.

whereas BGP_{inf} and BGP have 20%/22% and 37%/47% more, respectively, thanks to the GPU-context appropriate prediction algorithm (Section 5.2.3).

In general, the CG-only scheme falls short by significantly overfetching data for FG-leaning applications while the FG-only scheme disrupts the memory access behavior of several benchmarks, despite its advantage in reducing off-chip traffic. A static GDU configuartion, preferrably matching application characteristics (e.g., Avg_{sec} provided by the profiler/autotuner), typically performs best in terms of overall bandwidth utilization, maximizing energy-efficiency. While less effective than the best-performing CG-/FG-only scheme for each application, dynamically-driven LAMAR approximates the characteristics of the static GDU schemes, balancing the benefits of CG-fetches while reducing traffic when feasible. Compared to BGP, SPP-based prediction lacks robustness and fails to effectively reduce off-chip traffic for SSSP, BFS1, SP,

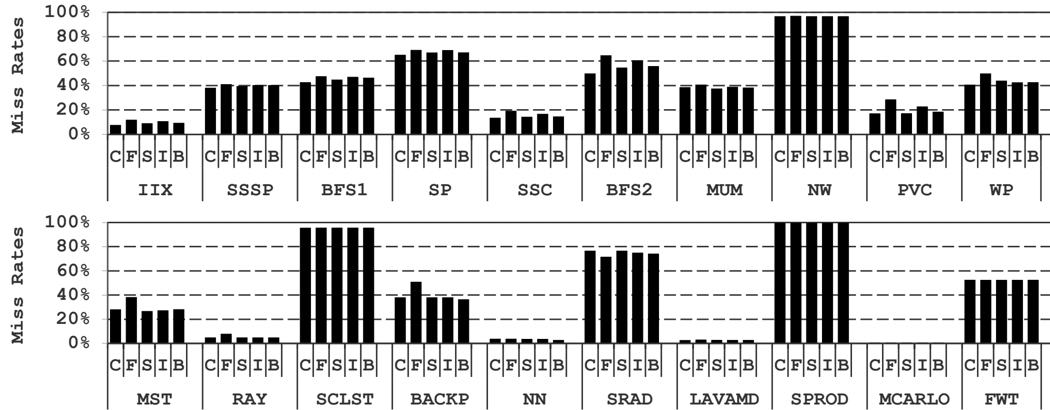


Figure 5.11: Changes in L2 cache miss rates with LAMAR.

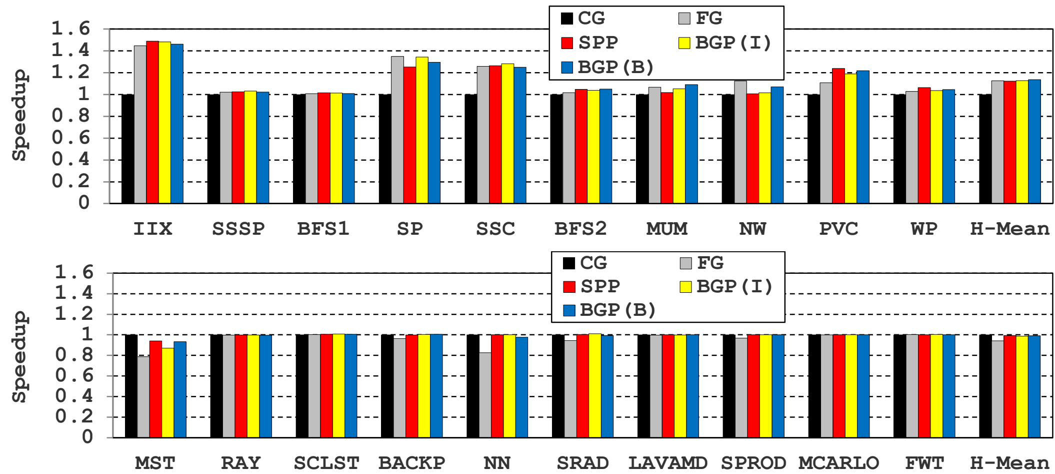


Figure 5.12: Normalized speedup. BGP(I) and BGP(B) represent BGP_{inf} and BGP.

MUM and NW. However, SPP is still advantageous compared to a static memory hierarchy.

5.3.2.2 Overall Performance

Figure 5.12 shows the overall speedup from adopting LAMAR memory schemes. In general, all LAMAR predictors provide significant benefit over the conventional CG-only memory system while executing FG-leaning applications thanks to more efficient utilization of the off-chip bandwidth, demonstrating a maximum 49% boost and an average 12–14% improvement in performance. Note that applications with low average byte traffic (Figure 5.9) are still able to gain good performance improvements with LAMAR (e.g., PVC, SSC). Depending on the application characteristics and memory access behavior, there can be significantly high memory access intensity within a short time window (in which LAMAR can help resolve the performance bottleneck at the memory channels) even though the absolute volume of data sent through the memory channel is low on average.

LAMAR predictors also provide comparable performance to the CG-only scheme in executing CG-leaning applications—the biggest degradation is for MST (whose L1 caching efficiency is disrupted by LAMAR, lowering the IPC by 13% with BGP_{inf}). The static FG-only scheme adversely impacts 5 of the CG-leaning benchmarks, ranging from 4% (SPROD) to 22% (MST) performance degradation.

5.3.2.3 Impact on DRAM Power Efficiency

Correctly predicted FG-fetches reduce the number of read and write commands issued to DRAM. However, CG-fetches have the advantage of lever-

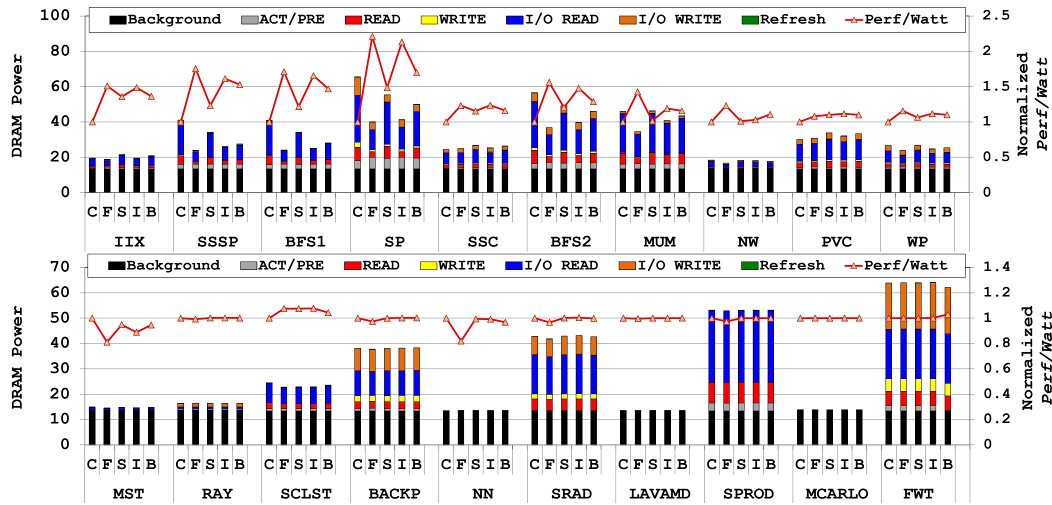


Figure 5.13: A breakdown of DRAM power (Watt, left axis) and the corresponding *Perf/Watt* (normalized to *CG*-only, right axis).

aging DRAM bank row locality by only having to open the corresponding bank row once. This is not the case for mispredicted *FG*-fetches (*NPRED_REF*), which require re-opening the bank row at a later time and lead to additional activate/precharge (ACT/PRE) commands. Figure 5.13 illustrates how the reduction in off-chip traffic correlates with DRAM power consumption. Overall, the benefit of reduced read/write commands outweighs the overhead of increased ACT/PRE commands. For *FG*-leaning applications, *FG*-only achieves the largest average power reduction of 19% (max 42%) while SPP/BGP_{inf}/*BGP* obtain an average 1%/13%/8% reduction (max 16%/39%/33%), respectively. Despite its low implementation cost, *BGP* is competitive with BGP_{inf} and outweighs SPP both in power reduction as well as Performance/Watt (*Perf/Watt*), with an average 27% increase in *Perf/Watt* while SPP and BGP_{inf} achieve an average 16%/34% improvement, respectively.

For *CG*-leaning applications, all LAMAR predictors perform comparable to the *CG*-only scheme whereas *FG*-only suffers from an average 5% degradation in *Perf/Watt* (maximum 20% degradation).

5.3.2.4 System-Level Power Efficiency

The system-level efficiency of different memory schemes is evaluated by combining the DRAM power model (Section 5.3.2.3) with the IPC-based GPU processor power model (Section 5.3.1). Recent literature [53, 34] estimates that the memory system consumes approximately 5 to 45% of the overall GPU power, depending on the application. LAMAR mainly improves the energy-efficiency of the memory hierarchy, so the overall improvement in *Perf/Watt* is less pronounced than its DRAM counterpart. Among *FG*-leaning applications, BGP_{inf} and *BGP* obtain an average 18% and 17% improvement in *Perf/Watt*, respectively. SPP helps the least among LAMAR predictors with an average 13% improvement in *Perf/Watt*. The *FG*-only mechanism, while achieving the highest average *Perf/Watt* improvement (19%), struggles in executing *CG*-leaning applications and significantly degrades *Perf/Watt* for MST, NN, and SRAD.

5.3.2.5 Sensitivity Study

This section summarizes LAMAR's sensitivity to key parameters. For conciseness, we primarily focus on the reduction in off-chip byte traffic for *FG*-leaning applications since this is where LAMAR gains most of its benefits.

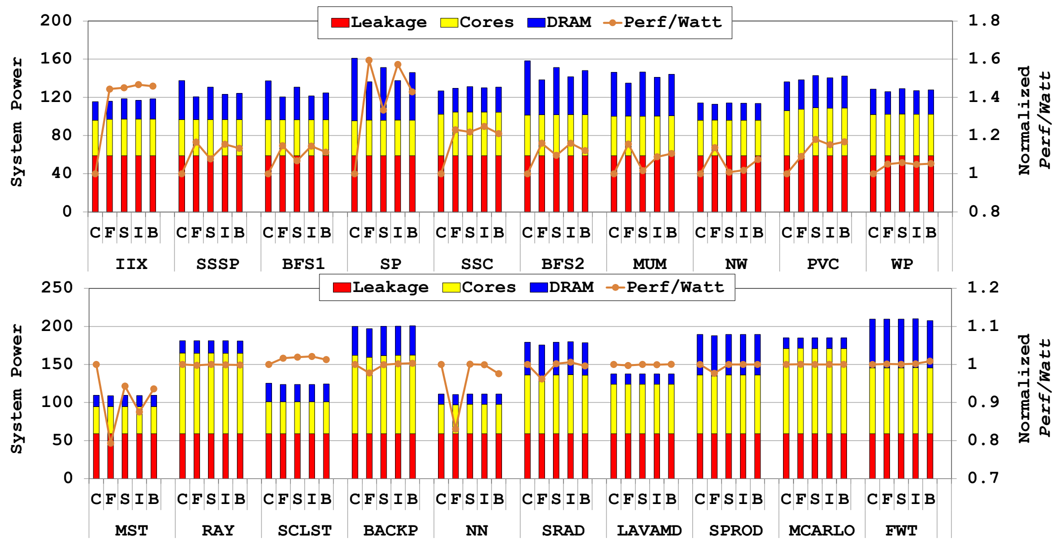
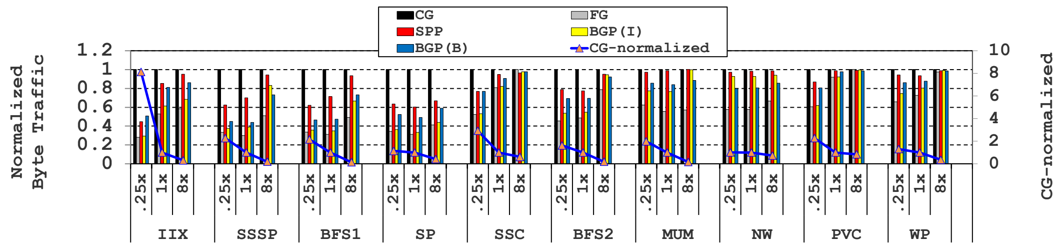


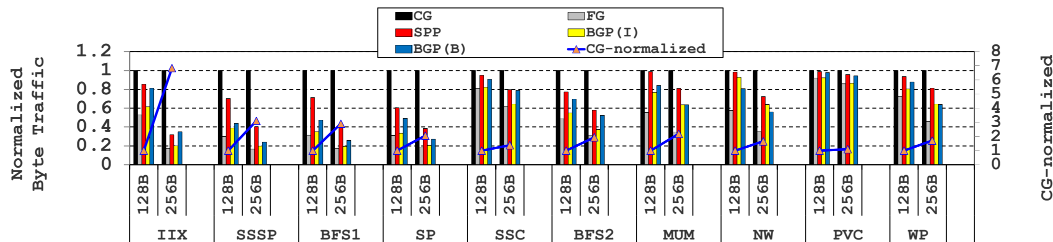
Figure 5.14: The system power consumption (Watt, left axis) and the corresponding *Perf/Watt* (normalized to *CG*-only, right axis). Note that the range of the upper plot is 0.8–1.8, whereas it is only 0.7–1.2 for the bottom plot.

Cache Capacity. As shown in Figure 5.15(a), DRAM traffic is generally reduced with larger on-chip caches (and vice versa for smaller caches) thanks to better caching efficiency. The benefits of LAMAR are still maintained across all *FG*-leaning applications and the relative reduction in traffic (compared to each configuration's *CG*-only scheme) is more pronounced with smaller caches (e.g., IIX, BFS1, SSC, BFS2, WP). *BGP*, for instance, exhibits an average 37%/33%/17% reduction in traffic with the three cache size configurations.

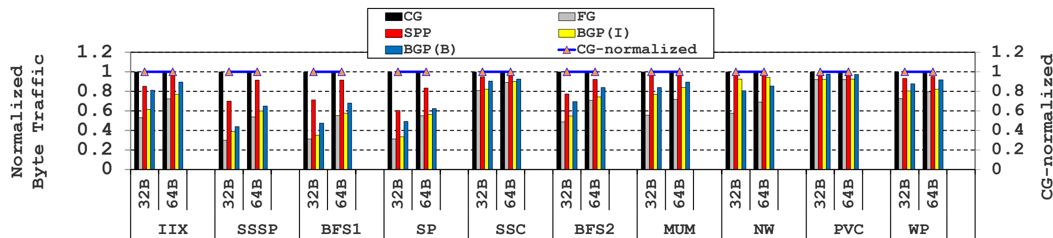
Cache Block Size. With a larger 256 bytes L2 cache block (256B), the baseline *CG*-only scheme is likely to overfetch even more off-chip sectors and to suffer from severe bandwidth under-utilization. Such behavior is illustrated in Figure 5.15(b) where a 256B configuration of *CG*-only uses an average 96%



(a) Sensitivity of LAMAR to reduced (0.25 times) and increased (8 times) L1/L2 capacity.



(b) Sensitivity of LAMAR to L2 cache block size. The overall L2 cache capacity is maintained equal to the baseline.



(c) Sensitivity of LAMAR to the minimum access granularity (32 bytes/64 bytes). Note that the L1/L2 cache capacity and cache block size are maintained equal to the baseline.

Figure 5.15: Sensitivity of off-chip traffic when varying (a) cache capacity, (b) L2 block size, and (c) minimum access granularity. The left axis represents the off-chip traffic of five configurations, normalized to each configuration's *CG*-only scheme. The right axis is used to compare the traffic of each configuration's *CG*-only scheme (*CG-normalized*) and is normalized to baseline.

more memory traffic than the baseline. The benefits of LAMAR, accordingly, are much more evident under the 256B configuration, where *BGP* reduces off-

chip traffic by an average of 58% compared to the 33% reduction using the baseline cache block size.

Larger Sector Size. To demonstrate the benefits of LAMAR with minimal changes to the GPU system, the proposed mechanisms without a sub-ranked off-chip memory system are also studied. A conventional GDDR5-based memory system provides a minimum access granularity of 64 bytes (Figure 2.4), so we evaluate LAMAR with a 64 bytes sector size and minimum access granularity. As depicted in Figure 5.15(c), the benefit of LAMAR is reduced from an average 33% traffic reduction to 20% with *BGP* due to the lack of sub-ranking and less fine control of data fetching granularity.

Thread-Level Parallelism. Recent literature [86, 35] shows that throughput processors make poor use of data caches, due to the high cache access intensity and the resulting low per-thread cache capacity. To this end, previous work makes the warp scheduler *cache conscious* [35] such that the number of warps able to access the cache are dynamically reduced if the cache is thrashing (hence throttling thread-level parallelism [TLP] available at the GPU core). Such cache-conscious warp scheduling (CCWS) is therefore only effective when the application is both cache-sensitive and is thrashing. While LAMAR focuses on wasted transfers due to granularity mismatches in the system and is orthogonal to CCWS, we nonetheless evaluate the effectiveness of LAMAR on top of this technique. Since CCWS is effectively a dynamic mechanism that

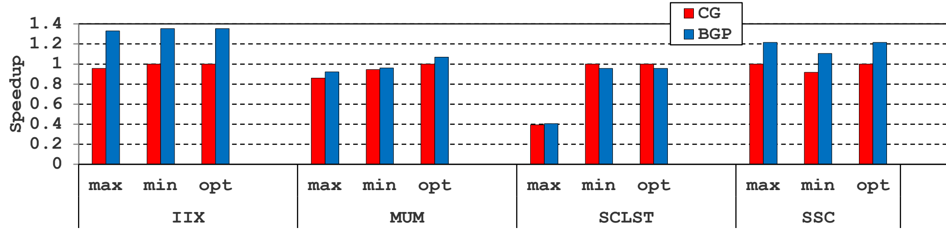


Figure 5.16: Sensitivity of LAMAR to available TLP. We experiment by sweeping through the number of schedulable CTAs within a GPU core from one (*min*) to its maximum allowable number (*max*) – which is limited by available hardware resources. The speedup with the optimal number of CTAs is reported as *opt*. The figure shows the IPC normalized to the *CG*-only scheme with *opt* TLP. Only three applications (IIX, MUM, SCLST) benefit from TLP throttling, meaning that, the best performance is generally achieved with the maximum level of TLP. Such common-case application behavior is represented by SSC whose performance is degraded with reduced TLP.

approximates the statically chosen optimal level of TLP, we experiment with LAMAR on top of CCWS as detailed in Figure 5.16. As depicted, three of the 20 applications we study (IIX, MUM, SCLST) benefit from TLP throttling and LAMAR remains effective in the presence of TLP tuning.

Miscellaneous. As mentioned in Section 5.2.3, the prediction quality of the baseline *BGP* microarchitecture is relatively robust with bit-array sizes larger than 2K bits. Performance is improved from 2% to 7% with a 4K bit-array, but saturates when going above 4K bits. Changing TH_{FG} and $SKEW_{thres}$ (Figure 5.6) also affects off-chip traffic and performance, but overall trends remain similar to the analysis discussed throughout this section (so long as $SKEW_{thres}$ is above 0.7 and TH_{FG} is less than three sectors).

5.3.2.6 Implementation Overhead

LAMAR is implemented using a sector cache and a sub-ranked memory system, the overheads of which are well established in previous literature [75, 76, 77, 78, 79, 80]. In addition, each cache partition is augmented with a GDU. Static GDU configurations require no additional hardware, but necessitate profiler/autotuner support to provide recommended granularity information. Further exploration of identifying the optimal granularity at compile time is left for future work.

For dynamic GDU schemes, the proposed *BGP* microarchitecture (using a dual bit-array bloom filter) requires: (1) $4K$ bits of storage per GDU, (2) 6 sets of *XOR* logic gates for the hash functions, and (3) control logic to insert/test the membership of the bloom filter (Table 5.2). Given that the 32-entry prediction table used for CAPRI consumes less than 10mW, despite its fully-associative structure with 1088 bits of storage (Section 4.4.2.6), we estimate that each GDU consumes less than 40mW.

5.4 Discussion

This section presents several discussion points related to LAMAR.

5.4.1 LAMAR on Future Memory Technologies

In order to be concrete and to allow a detailed evaluation, this work leverages GDDR5 as the main memory technology. Future GPUs, however, may use evolving memory interface standards that utilize 3D packaging tech-

nology, such as the *Hybrid Memory Cube* (HMC) [95] or *High-Bandwidth Memory* (HBM) [96]. Although these interfaces are likely to offer much higher bandwidth than GDDR5, GPU arithmetic performance will increase as well, and effectively utilizing memory throughput will remain critical to performance and efficiency. In fact, capacity-to-bandwidth ratio is likely to increase with the use of HMC packages—this implies that bandwidth utilization will increase in importance, amplifying the potential benefits of LAMAR. The proposed access granularity for these interfaces is also similar to that of GDDR5 devices today, with HMC proposing an access granularity of $32 - 256$ bytes and with HBM likely to use a 32 bytes granularity similar to the WideIO standard [97]. Thus, the opportunity and policies we propose for LAMAR should generally apply equally well.

With 3D packaging, it is likely that the memory controller will be partitioned between the processor and the DRAM die stacks: scheduling is likely to remain close to the processor, where knowledge of priorities and requests is readily available, while implementation of the DRAM access protocol will be relegated to the controller within each stack [95]. This partitioning will require a re-design of how LAMAR controls the memory modules (owing to the fact that sub-ranks are essentially internalized and hidden within each stack). Because scheduling is still delegated to the processor, LAMAR will have to be modified to account for sending the appropriate request packets to maximize transfer efficiency. Exploring LAMAR under such designs is left for future work.

5.4.2 Error Correction

Current GPUs support error correction with error correcting codes (ECC). While the details of the memory protection schemes in industry are not publicly known, one way of flexibly supporting error correction without dedicated DRAM chips is through virtualized ECC [98]. The approach taken by LAMAR is amenable to such error correction, in a similar manner to prior work [74, 99].

5.4.3 Alternative FG Cache Management

LAMAR uses a simple sector cache to manage *FG* data in the on-chip cache hierarchy, as our current study focuses on the efficient management of off-chip data for irregular GPU applications. However, alternative *FG* cache management schemes exist, such as the decoupled sectored cache [100], pool-of-sectors cache [101], or the spatio/temporal cache [102]. Such more advanced cache architectures could be adopted to increase the *effective capacity* of the cache for irregular applications. Some irregular applications are very sensitive to the (typically limited) on-chip storage capacity, such that these alternative caches could significantly increase performance.

5.4.4 Other Dynamic Bloom Filter Mechanisms

The *BGP* incorporates two temporally-separated bloom filters to support the aging of membership data and to allow space-efficient operation with a dynamic stream of accesses. The temporal aging of bloom filter entries for

dynamic data has been addressed by Deng and Rafiei [103] by associating a slowly degrading count with each storage cell. However, this design makes inefficient use of storage and is unlikely to perform competitively with the *BGP*. The concept of maintaining and swapping two temporally-separated bloom filters has been previously employed in software for filtering dynamic data [104, 105, 106]. *BGP* is the first application of such a scheme to memory access granularity prediction in hardware and is unique in its implementation and default prediction inverting algorithm. Yoon [107] recently proposed an alternate two-buffer algorithm for filtering dynamic data that could provide modest accuracy benefits for the *BGP*.

5.4.5 Impact on Programmability

Maximizing off-chip memory bandwidth utilization necessitates that the programmer carefully align/confine the intra-warp, per-thread memory access to within a single cache block, so that memory divergence is minimized and memory access overhead is amortized across the warp. As discussed in Section 4.5.2, requiring programmers to understand and consider such low level microarchitectural characteristics substantially hampers programmer productivity. LAMAR allows programmers to be less concerned about such low level details as the bi-modal granularity predictor will automatically, and transparently evaluate the memory access granularity and seek to minimize waste in bandwidth utilization when fine-grained requests are made. Such support improves the programmability of the SIMT execution model.

5.5 Summary

The increasing popularity of general-purpose GPU programming and the growing irregularity of some throughput-oriented programs necessitate a fine-grained GPU memory system. Meanwhile, the continuing need for the high-performance acceleration of regular, well structured programs and graphical workloads makes coarse-grained memory accesses compulsory as well. I propose LAMAR, an adaptive and reactive hardware-only memory scheme for GPUs and throughput-oriented processors that achieves superior efficiency across a range of general-purpose GPU applications. By dynamically predicting the temporal and spatial locality of memory accesses, LAMAR mitigates the deficiencies of static-granularity memory systems and prior mixed-granularity memory schemes for control-intensive CPUs. In addition, the hardware required for LAMAR is simple and non-intrusive enough to be readily implemented in a many-core GPU and its adoption requires no programmer intervention. Results show that LAMAR provides an average 14% increase in performance (max 49%), 33% reduction in average off-chip traffic (max 64%), and an average 17% improvement in system-level energy-efficiency (max 47%).

Chapter 6

Future Research Directions

The previous chapters discussed the irregularity of SIMT control and memory accesses, their implication in a massively multi-threaded computing environment, and how the mechanisms I developed address these challenges in a cost-effective manner. There are, however, still several opportunities for future investigation that can extend the topics discussed in this dissertation. This chapter provides discussion points that will motivate such future research work.

6.1 A QoS-Aware Throughput Processor Architecture

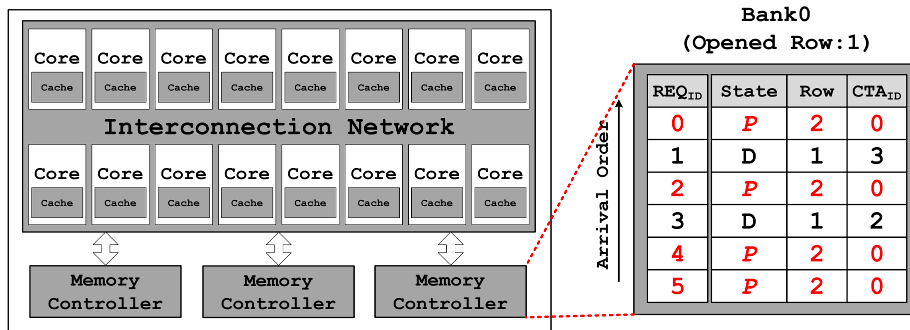
Motivation. As proven in chip-multiprocessor (CMP) systems [108, 109], granting different (or equal) levels of access priority to shared resources is vital for maximizing overall system throughput. In contrast to CMPs, throughput processors execute a massive number of, essentially identical instances (threads) of a single program. As a result, the QoS each thread obtains has been ignored by academia until very recently; previous literature discussing a (GPU-style) throughput processor mostly assumes that all threads have *equal* access priority to the shared resources (e.g., warp scheduler, on-chip caches,

interconnection network, DRAM). Due to the significant number of concurrently executing threads, however, such equal sharing of resources results in threads potentially interfering with one another, degrading performance. As detailed in Section 5.1.2, the on-chip data caches, for example, are likely to suffer from thrashing when a large number of threads with irregular memory access characteristics concurrently fight over the limited cache capacity. Because the likelihood of thrashing is correlated with the number of threads timesharing the data cache, it is fundamentally challenging for throughput processors to enable *all* threads to effectively use the cache. Recent work by Rogers et al. [35] proposed a warp scheduling mechanism that throttles the execution of a subset of threads, such that cache contention is alleviated. Kayiran et al. [110] introduced a similar approach that adaptively throttles the number of CTAs that can execute. While these proposals provide good insights on the importance of QoS adjustment across different threads, they leave other system resources underutilized, such as thread issue bandwidth, interconnection bandwidth, and memory bandwidth. Also, given that throughput processors leverage abundant TLP as a way of achieving high latency tolerance, such throttling of thread execution is only effective when the benefit of higher caching efficiency outweighs the reduction in latency tolerance (as detailed in Section 5.3.2.5).

Proposed Approach. State-of-the-art warp scheduling mechanisms [24, 35, 36] already adopt a priority-based scheduling policy where a subset of warps

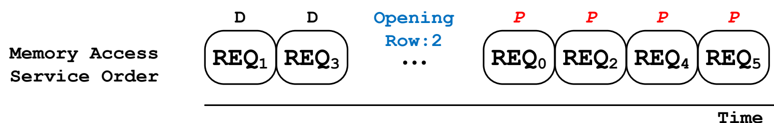
receive preferential scheduling decisions compared to other warps – which is intended to overlap off-chip memory transactions and maximize memory-level parallelism. I argue that threads that are temporally prioritized in one component (e.g., warp scheduler, on-chip caches) should also receive higher QoS across the entire system architecture (e.g., interconnection, DRAM accesses, etc). A *QoS-aware/-adaptive* throughput processor architecture seeks to *accelerate and better localize* the servicing of computation and memory requests from *prioritized* threads (warps/thread-blocks). By providing high QoS to a subset of concurrently executing threads, those that are prioritized are given the illusion that they *solely* have access to the entire set of shared resources, alleviating interference with the other threads. The key insight is that the overall architecture is driven in a way that minimizes the resident-time (time to complete execution) of prioritized threads, even if it inevitably leads to throttling the progress of deprioritized threads. Once prioritized threads complete, previously deprioritized threads receive higher QoS. In essence, prioritized threads are accelerated while resident deprioritized threads provide latency tolerance for maximal efficiency.

Note that achieving the above goal is difficult with a local, static QoS policy (e.g., round-robin warp scheduling [24, 36], equal access-priority to the cache, shared interconnection channel, *first-row, first-come/first-serve* (FR-FCFS) [111] policy in DRAM schdeuling). The FR-FCFS based DRAM scheduler, for instance, also dilutes the collaborative QoS-approach as illustrated in Figure 6.1(c). Here, prioritized accesses are not given higher QoS because

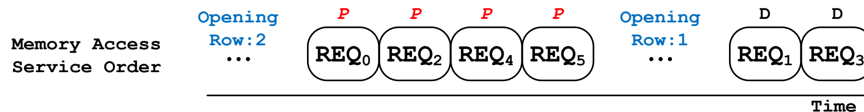


(a) Target components (gray-colored) of the QoS-aware throughput processor

(b) Example memory access sequence



(c) Memory service ordering when using an uncoordinated scheduler (FR-FCFS)



(d) Memory service ordering when using a coordinated scheduler that prioritizes *P*-accesses

Figure 6.1: Example showing the importance of coordinating *prioritized* (P) memory accesses in relation with the *deprioritized* (D) accesses. Figure assumes the warp scheduler prioritizes scheduling from *oldest* thread-blocks (e.g., memory accesses from CTA_{ID}=0). Note that the D-accesses are serviced earlier than the P-accesses with the (c) FR-FCFS policy. The (d) *FP*-FRFCFS policy can prioritize the P-accesses at the cost of opening one additional row.

FR-FCFS favors opened-row accesses and do not differentiate between prioritized/deprioritized accesses. I envison that a QoS-aware DRAM scheduler can be developed to have the scheduler take into account the different access priorities among prioritized/deprioritized accesses. By having prioritized

memory accesses given higher scheduling priority than the open-row accesses (Figure 6.1(d)), the progression rate of prioritized threads can be accelerated even further. Example scheduling policies that are QoS-aware include 1) FRFP-FCFS (among the FR, *first-prioritized*), 2) FP-FRFCFS (among the FP, FR), and 3) some variation of the above that is dynamically adjusted based on different phases of the program.

Designing and evaluating the aforementioned QoS-aware throughput processor architecture is beyond the scope of this dissertation. In order to demonstrate the potential of a QoS-aware throughput processor architecture, this section provides a case study on a *priority-based cache allocation* (PCA) scheme, which incorporates the notion of QoS on throughput processor cache hierarchies.

6.2 Case Study: Priority-Based Cache Allocation

As mentioned before, prior work proposed variants of TLP throttling to reduce cache contention and improve performance. However, throttling approaches can either fail to reduce the thread working set enough to make best use of the shared on-chip resources or underutilize register file thread contexts and off-chip memory bandwidth.

6.2.1 Motivation and Key Insights.

The key objective of PCA is to improve cache hit rates and block reuse, but also utilize other shared resouces more effectively by minimizing

the amount of TLP throttling. PCA categorizes threads into three categories: (a) those that can execute and allocate space inside the on-chip cache (prioritized threads), (b) those that can execute but cannot allocate space on-chip (deprioritized threads, which will *bypass* the cache fill process), and (c) those that are completely throttled and cannot execute (throttled threads). Because deprioritized threads are not allowed to allocate cache blocks, the working set of the prioritized threads is less interfered, minimizing cache pollution. At the same time, deprioritized threads are still able to execute (but at slower rate compared to prioritized threads), enabling better utilization of system shared resources and overcoming the deficiencies of total thread-throttling schemes [35, 110].

6.2.2 PCA Implementation.

Categorizing threads into the aforementioned three types is achieved by handing out *tokens*. Tokens represent priority across the cache hierarchy and indicate privilege to allocate space inside the on-chip cache. When the warp scheduler issues threads to execute, a subset of the active threads (warps) receives the tokens and those threads are prioritized while others are deprioritized, or throttled. There exists a huge design space in terms of determining the types of tokens, the optimal number of available tokens, the token assignment/release policy, the cache block allocation/eviction policy, etc .... This case study therefore assumes the following in order to sketch one possible implementation of the PCA scheme while highlighting its potential.

Token Types. The example implementation of PCA assumes that the tokens are only used for (de)prioritizing cache accesses at the L1 cache, which is one of the most constrained resources on chip (Table 5.1). Cache accesses at the shared last-level L2 caches are handled identically as in the baseline cache model, regardless of the access priority determined by (L1-only) tokens. Note that other token variations are also possible and will be interesting to explore (discussed in Section 6.2.4).

Token Count. This study assumes that the number of tokens available within each GPU core (which will be handed out by the warp scheduler) are statically designated on a per-kernel basis, either by the programmer or by an autotuner/profiler. In Section 6.2.3, we exhaustively simulate all possible combination of tokens and report the results under the best performing token count. The number of threads that will be throttled from execution will also be determined similarly, the methodology of which is detailed in Section 6.2.3.3 (Figure 6.3).

Token Assignment Policy. The baseline warp scheduling policy we adopt is *greedy-then-oldest* (GTO) [35] where the most recently scheduled warp (the warp that gets greedily prioritized over others) and then the oldest warps receive the highest priority. This study observes that tightly coupling the token assignment policy with the warp scheduling mechanism provides high performance as it tends to better localize and accelerate the prioritized warps

execution rate. This case study therefore configures the PCA token assignment policy as assigning N tokens to the N oldest warps. Having the token assignment policy configured identically to the GTO warp scheduler performed worse than assigning tokens to the oldest warps; this is mainly because compared to the oldest warps, the most recently scheduled warp tends to be less deterministic, so greedily prioritizing the most recently scheduled warp tends to disrupt locality.

Token Release Policy. A prioritized warp retains the token until it terminates execution. Once the prioritized warp terminates, the next oldest warp (which currently does not hold a token) is assigned the token. This is in order to correlate the warp scheduling policy with the token assignment and release policy, as the chosen warp scheduling policy already prioritizes oldest warps until termination.

Cache Block Allocation Policy. Cache accesses of prioritized warps are always *guaranteed* to allocate space such that their working sets are cache resident. Non-token holders, on the other hand, can only allocate space if there exists a vacant slot within the corresponding cache set. If all the cache blocks within the cache set are fully occupied by prioritized warps, then the cache fill-process is *bypassed* in order to avoid evicting prioritized cache blocks. Fill-bypassed memory requests still allocate miss-status holding register (MSHR) entries [112] in order to guarantee the baseline weak memory consistency model of current GPU architectures [28].

Table 6.1: Simulator configuration for PCA evaluation.

Number of GPU cores	15
Threads per GPU core	1536
Threads per warp	32
SIMD lane width	32
Registers per GPU core	32768
Shared memory per GPU core	48KB
Warp scheduling policy	Greedy-then-oldest [35]
L1 cache (size/associativity/block size)	16KB/4-way/128B
L2 cache (size/associativity/block size)	768KB/16-way/128B
Memory bandwidth	177.6 GB/s
Memory controller	Out-of-order (FR-FCFS)

Cache Block Eviction Policy. Similar to the allocation policy, this study assumes that possessing a token indicates a warp has permission to initiate cache block eviction whereas non-token holders are not allowed to perform replacements, unless the victim block is allocated by a deprioritized access.

6.2.3 PCA Evaluation and Analysis

This section presents the simulation methodology followed by a detailed evaluation of PCA.

6.2.3.1 Methodology

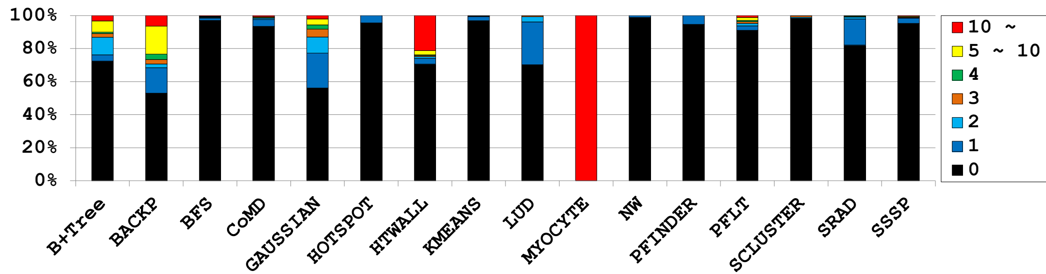
PCA has been modeled using GPGPU-Sim [46, 47] and the overall GPU architecture has been configured similar to NVIDIA's GTX480 [38] using the configuration file provided with GPGPU-Sim [48]. Key microarchitectural parameters of the baseline configuration are summarized in Table 6.1.

Table 6.2: Benchmarks studied for PCA evaluation.

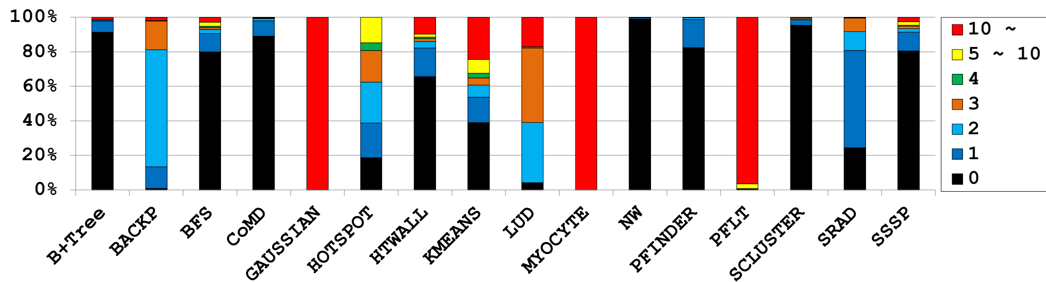
Abbreviation	Description	#Instr.	Ref.
B+Tree	B+ Tree	645M	[14]
BACKP	Back propagation	190M	[14]
BFS	Breadth first search	2.8B	[16]
CoMD	Molecular Dynamics	6.3B	[113]
GAUSSIAN	Gaussian elimination	230M	[14]
HOTSPOT	Hotspot	550M	[14]
HTWALL	Heartwall	8.8B	[14]
KMEANS	K-means	719M	[35]
LUD	LU decomposition	302M	[14]
MYOCYTE	Myocyte	1.2M	[14]
NW	Needleman-Wunsch	207M	[14]
PFINDER	Pathfinder	649M	[14]
PFLT	Particle-filter	118M	[14]
SCLUSTER	Streamcluster	343M	[14]
SRAD	SRAD	2.4B	[14]
SSSP	Single-source shortest paths	696M	[14]

6.2.3.2 Benchmarks

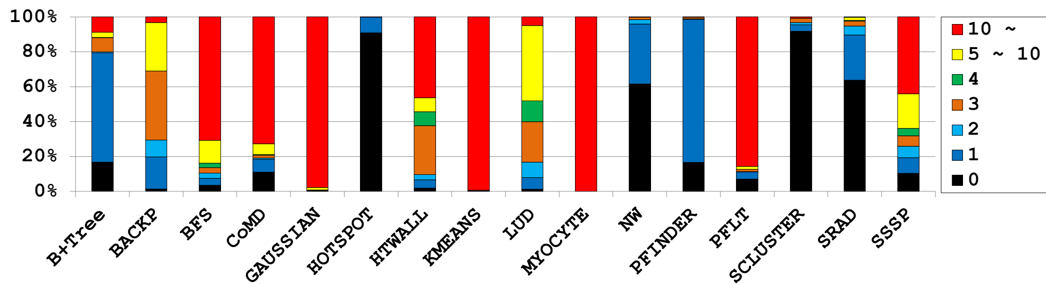
PCA is evaluated with applications from Rodinia [14], LonestarGPU [16], and CoMD [113] (Table 6.2). All applications have been executed to completion, with the exception of SSSP; due to long simulation periods, SSSP is executed only up to the point where overall IPC exhibits small variation among different iterations of the kernel. Among the 16 applications in Table 6.2, four of them (BFS, CoMD, KMEANS, and SSSP) were shown to exhibit better performance with a larger cache. The reason for such varying cache-sensitivity is twofold: (a) the application's dataset is either streaming or exhibits low cache block reuse even with an *ideal* large cache (Figure 6.2(c)), or (b) the baseline cache hierarchy can capture the working set relatively well such that better



(a) Temporal reuse at the L1 cache.



(b) Temporal reuse at the L2 cache.



(c) Temporal reuse with an *ideal* cache with infinite capacity

Figure 6.2: The distribution of repeated accesses to cache blocks in the (a) L1 cache, (b) L2 cache, and (c) an ideal L1 cache with infinite capacity (no evictions once cached on-chip).

caching efficiency provides little benefit. This case study therefore focuses on the four highly cache-sensitive applications to highlight the benefits of PCA.

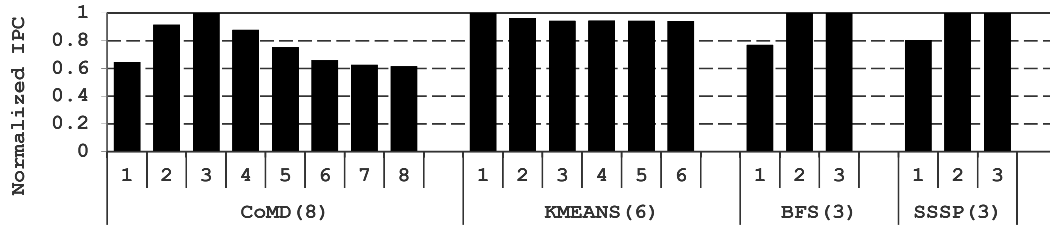


Figure 6.3: Normalized IPC while limiting the number of CTAs available for scheduling. Numbers next to the names of each benchmark represent maximum number of CTAs able to be allocated within each GPU core. Each benchmark has therefore been executed with having one to maximum CTAs available for scheduling.

6.2.3.3 TLP Sensitivity

As discussed in Section 5.3.2.5, previous TLP throttling mechanisms [35, 110] seek to dynamically approximate the statically chosen optimal level of TLP within each GPU core. In order to compare the benefits of PCA with such total TLP-throttling mechanism, the four cache-sensitive benchmarks are executed with varying levels of available TLP. Figure 6.3 shows how the overall performance varies with different number of CTAs able to be executed within a GPU core. For some applications (CoMD and KMEANS), running with the maximum TLP that the hardware can support does not result in the best throughput. CoMD, for instance, performs best with only 3 of 8 CTAs enabled. The performance of other applications (BFS and SSSP), on the other hand, does not improve with TLP throttling because the reduction in latency tolerance outweighs the benefits of better caching efficiency (e.g., best performance is achieved with maximum TLP). Based on these results, we refer to the baseline GPU executed with *maximum* feasible CTAs as `Max-CTA`. GPUs

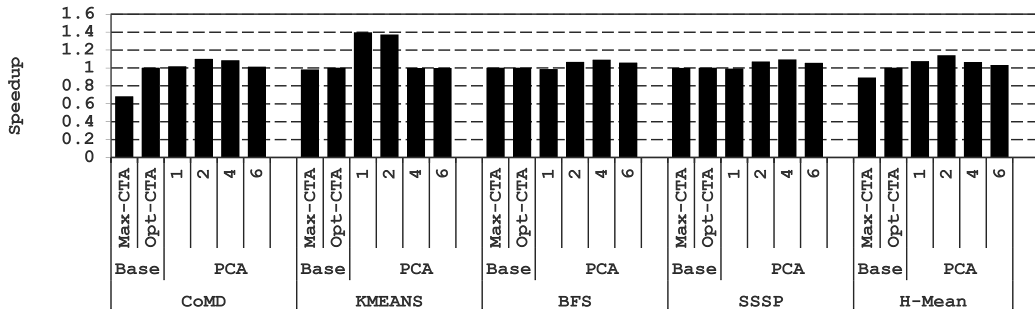


Figure 6.4: Speedup with PCA. `Max-CTA` designates execution with *maximum* number of active CTAs. `Opt-CTA` represents statically throttled version with *optimal* number of active CTAs (which leads to best performance) determined by Figure 6.3. 1/2/4/6 represents number of tokens available for each warp scheduler. Note that each GPU core contains two warp schedulers [38, 46], so the number of available tokens within a core are 2/4/8/12.

configured with the *optimal* number of available CTAs (using Figure 6.3, for best performance) will be referred to as `Opt-CTA` and is used as baseline to compare against PCA. In addition, the optimal number of CTAs to throttle for `Opt-CTA` will also be leveraged by PCA to determine the number of throttled threads. The prioritized/deprioritized threads will be determined by the programmer-specified token counts which is determined by exhaustively sweeping through all possible token counts as detailed below.

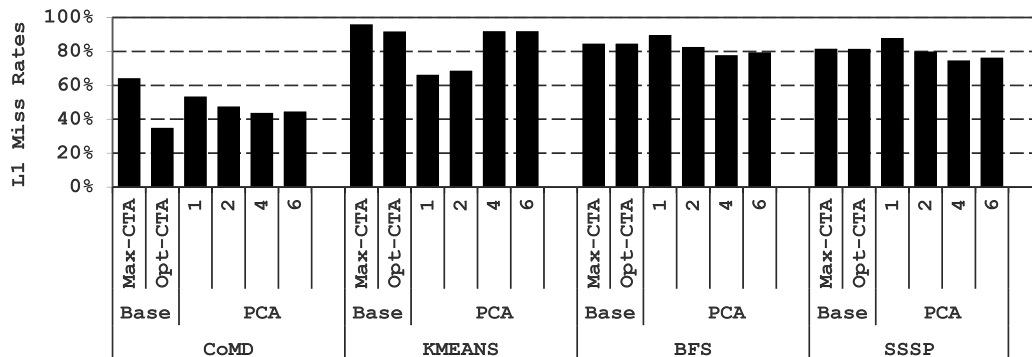
6.2.3.4 Overall Performance

This section evaluates the effectiveness of PCA in terms of performance improvements (compared to `Opt-CTA`). Overall, PCA with two tokens provides the highest average (harmonic means) speedup with 14% enhancements but the optimal token count varies depending on application characteristics (Fig-

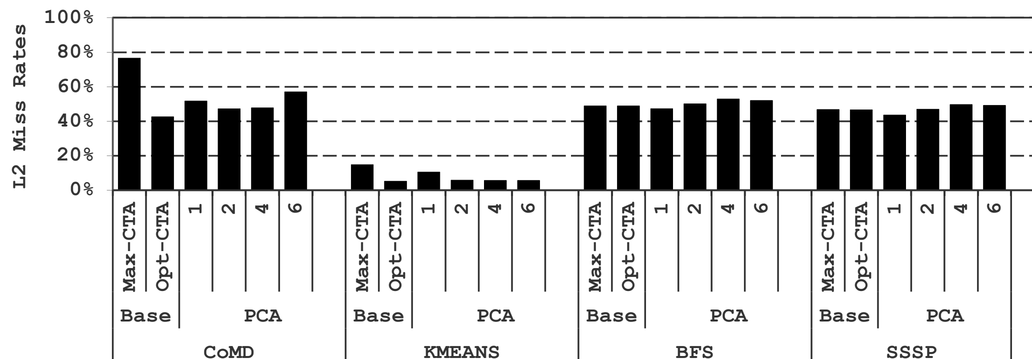
ure 6.4). CoMD, BFS, and SSSP achieve 10%, 8%, and 9% performance improvement respectively, and KMEANS exhibits the highest benefits with 39% speedup. Note that PCA provides further benefits to BFS and SSSP, which do not benefit much from total TLP-throttling techniques [35, 110]. BFS and SSSP are sensitive to TLP reduction (Figure 6.3), so total TLP-throttling techniques fail to provide means to reduce cache contention. PCA, on the other hand, is able to provide maximum TLP to the warp scheduler while still being able to reduce cache contention and enhance bandwidth utilization thanks to the token-based caching mechanism. The improvements in caching efficiency, cache block reuse, and off-chip bandwidth utilization are detailed below.

6.2.3.5 Memory Access Efficiency

Caching Efficiency. Figure 6.5 and Figure 6.6 show the impact on cache miss rates as well as the average block reuse at L1/L2 caches. Comparing the results with Figure 6.4, in many cases the best performance does not necessarily correspond to the lowest miss rates nor the highest average block reuse. For instance, BFS and SSSP exhibit best performance with 4 tokens, but the highest average reuse is observed with only a single token assigned per warp scheduler. A small number of tokens means only a handful of warps can preserve its working set on chip (preventing other warps from allocating their working sets), which may not be optimal from the overall system perspective. Such a trend is exhibited by most of the benchmarks where having only a single



(a) L1 miss rate.

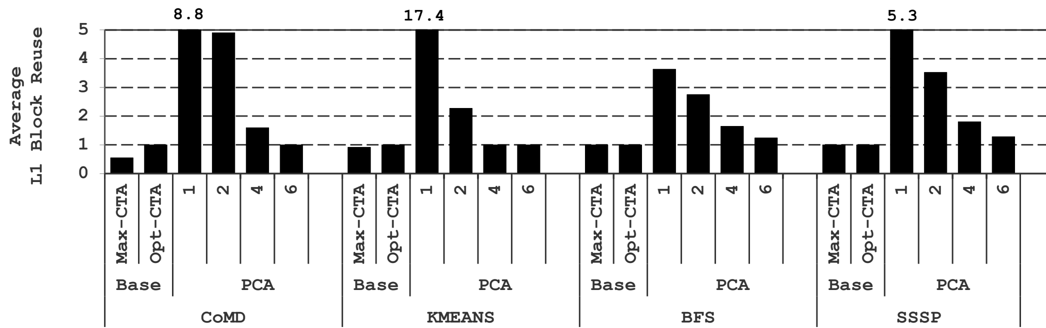


(b) L2 miss rate.

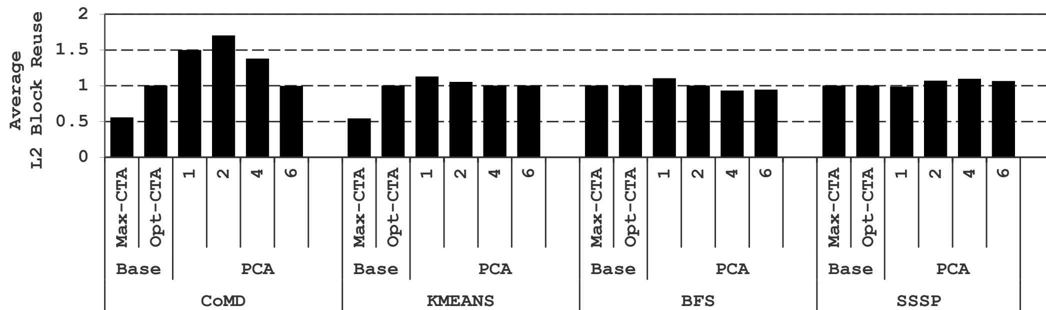
Figure 6.5: Changes in L1/L2 cache miss rates with PCA.

token available leads to noticeable increase in L1 cache miss rates, despite substantial improvements in average block reuse.

Overall, when the number of tokens available is optimally chosen, the caching efficiency and average block reuse are balanced, leading to best performance.



(a) Average L1 cache block reuse.



(b) Average L2 cache block reuse.

Figure 6.6: Changes in average L1/L2 cache block reuse with PCA.

Memory Bandwidth Utilization. Figure 6.7 shows the off-chip bandwidth utilization across different schemes. Because `Opt-CTA` generally reduces the number of threads to maximize shared resource utilization (except for BFS and SSSP where `Opt-CTA` is identical to `Max-CTA`), DRAM bandwidth utilization is noticeably reduced for CoMD and KMEANS (120% and 180% reduction). PCA on the other hand is able to significantly improve off-chip bandwidth utilization by 30% on average and a maximum of 103% for KMEANS. The improved bandwidth utilization is due to the non-token holders being able to make forward progress, which `Opt-CTA` completely throttles from execution.

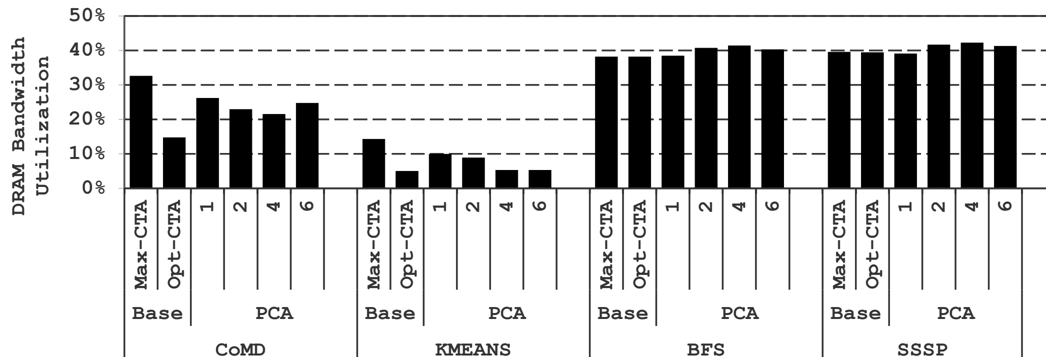


Figure 6.7: Changes in off-chip bandwidth utilization with PCA.

Such behavior, coupled with the previously discussed better caching efficiency, enables higher performance with PCA.

6.2.4 Discussion and Future Extensions

We have so far explored the potential benefits of a statically configured PCA under a throughput processor environment. This section summarizes some possible future extensions.

Dynamic PCA. Static PCA is an effective tool for expert programmers to tune the performance of GPU applications. By trying out different number of tokens, programmers can optimally configure an application for higher throughput. Finding the best number of tokens, however, requires the programmer to thoroughly understand the GPU microarchitecture. It is hence desirable for the programmer to be provided with a *dynamic* PCA mechanism where the optimal number of tokens is derived at runtime without programmer intervention. One possible way of implementing the dynamic PCA is to ex-

ploit the many-core nature of throughput processors: each GPU core tries out a different number of tokens and the global predictor examines which token configuration best suits application needs. Such global voting mechanism has been successfully deployed by Lee and Kim [114]. Exploration of such dynamic PCA scheme is left for future work.

Token Types. While this case study explores PCA with L1-only token configurations, other token variations are also possible: (a) L2-tokens, which will be assigned to warps mutually exclusive to the L1 token holders, (b) DRAM-tokens, which will be assigned to warps not assigned with L1/L2-tokens, and some combination of the above. Exploring such token variations in the context of PCA are left for future work.

Token Assignment/Release Policies. Current implementation of PCA hands out tokens to the oldest warps and is never shared nor transferred to other warps until termination. Such unfairness-oriented token assignment and release policy may not be desirable if an application were to contain different program phases (e.g., prioritized threads require on-chip cache space only during the initial phase of the program, which can prevent other deprioritized threads from using it). It will therefore be interesting to study different token assignment and release mechanisms that dynamically predict and adapt to the program phase behavior, such that on-chip caches are best utilized.

6.3 Summary

This chapter discussed potential future works and extensions that are relevant to the key theme of my dissertation. I proposed and discussed a QoS-aware throughput processor architecture that can potentially better manage the contention at shared resources. We explored a priority-based cache allocation scheme as a case study to highlight the potential benefits and the importance of QoS-awareness in a throughput computing environment. Future work is to extend the intuition provided by this case study across the whole system stack.

Chapter 7

Conclusions

As throughput processors, and specifically GPUs, gain momentum in being widely used for general purpose computing, efficient execution of irregular applications will be vital for their continued success. This dissertation presents and evaluates multiple performance optimization schemes for GPUs so that they can better manage irregularity. In the presence of SIMT control and memory access irregularity, I demonstrate that my proposed mechanisms can substantially improve thread-level parallelism, compute resource utilization, and off-chip memory bandwidth utilization in a cost-effective manner.

The key intuition behind the proposed mechanisms are: (a) exploiting thread-level parallelism within and across execution paths and (b) judiciously coordinating memory access granularity across the memory hierarchy. By adopting my proposed ideas, I believe throughput processors can manage irreguarity much more effectively, such that they can truly be considered for "general purpose" computing. Below is a summary of the key contributions of my dissertation.

Dual-Path Execution Model. I propose and evaluate the dual-path execution model in Chapter 3 to alleviate the SIMT path serialization problem.

The proposed mechanism can enhance thread-level parallelism by allowing the true and false paths of a divergent branch to be executed in an interleaved manner. Unlike previous proposals to this problem, my dual-path execution model requires no additional compiler support, requires minimal hardware extensions, yet is robust to changes in key microarchitectural parameters.

Compaction-Adequacy Prediction and SIMD Lane Permutation. In Chapter 4, I propose compaction-adequacy prediction and SIMD lane permutation as means to enhance the robustness and applicability of previous compaction-based throughput processor architectures. Previously studied solutions to the SIMD resource underutilization problem enforce needless synchronization of threads for dynamic formation of warps while only being applicable across a limited set of applications. My proposed ideas can intelligently filter out such redundant barrier overheads while enabling compaction to be effective across a much wider range of applications.

Locality-Aware Memory Hierarchy. Chapter 5 discussed the limitations of a coarse-grained memory hierarchy in a throughput computing environment. I observe that irregular memory accesses, combined with massive multithreading, often result in low temporal/spatial reuse of cache blocks due to the limited per thread cache capacity. I propose a locality-aware memory hierarchy that retains the advantages of coarse-grained accesses for high spatial/temporal data sets while minimizing useless overfetching by selective fine-grained accesses. By adaptively adjusting memory fetching granularity, the proposed

architecture can significantly reduce off-chip traffic and improve performance as well as energy-efficiency.

Bibliography

[1] NVIDIA Corporation. High Performance Computing: Accelerating Science with Tesla GPUs, 2013.

[2] NVIDIA Corporation. NVIDIA CUDA Zone, 2012.

[3] NVIDIA Corporation. CUDA C/C++ SDK CODE Samples, 2011.

[4] Panagiotis Vouzis and Nikolaos Sahinidis. GPU-BLAST: Using Graphics Processors to Accelerate Protein Sequence Alignment. In *Bioinformatics*, 2010.

[5] Steven Solomon, Ruppa Thulasiram, and Parimala Thulasiraman. Option Pricing on the GPU. In *12th International Conference on High Performance Computing and Communications*, 2010.

[6] Mark Harris. Fast Fluid Dynamics Simulation on the GPU. In *ACM SIGGRAPH*, 2005.

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy Sheaffer, and Kevin Skadron. A Performance Study of General-Purpose Applications on Graphics Processors using CUDA. In *Journal of Parallel and Distributed Computing*, 2008.

[8] Top500. http://www.top500.org.

[9] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. Introduction to Data Mining. Pearson Addison Wesley, 2005.

[10] Kirsten Hildrum and Philip Yu. Focused Community Discovery. In *International Conference on Data Mining*, 2005.

[11] Thomas COrmen, Charles Leiserson, and Ronald Rivest. Introduction to Algorithms. McGraw Hill, 2001.

[12] J. Barnes and P. Hut. A Hierarchical O(N log N) Force-Calculation Algorithm. In *Nature*, 1986.

[13] L. Paul Chew. Guaranteed-Quality Mesh Generation for Curved Surfaces. In *Proceedings of the Symposium on Computational Geometry (SCG)*, 1993.

[14] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization (IISWC-2009)*, October 2009.

[15] Shuai Che, Bradford Beckmann, Steven Reinhardt, and Kevin Skadron. Pannotia: Understanding Irregular GPGPU Graph Applications. In *IEEE International Symposium on Workload Characterization (IISWC-2013)*, 2013.

[16] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A Quantitative Study of Irregular Programs on GPUs. In *IEEE International Symposium on Workload Characterization (IISWC-2012)*, 2012.

[17] Pawan Harish and P.J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *High Performance Computing (HiPC)*, 2007.

[18] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P.J. Narayanan. Fast Minimum Spanning Tree for Large Graphs on the GPU. In *Proceedings of the Conference on High Performance Graphics (HPG)*, 2009.

[19] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU Graph Traversal. In *ACM SIGPLAN Symposium on Principles and Practive of Parallel Programming (PPoPP)*, 2012.

[20] Minsoo Rhu and Mattan Erez. The Dual-Path Execution Model for Efficient GPU Control Flow. In *19th International Symposium on High-Performance Computer Architecture (HPCA-19)*, February 2013.

[21] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *37th International Symposium on Computer Architecture (ISCA-37)*, 2010.

[22] Wilson W.L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow.

In *40th International Symposium on Microarchitecture (MICRO-40)*, December 2007.

[23] Wilson W. Fung and Tor M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *17th International Symposium on High Performance Computer Architecture (HPCA-17)*, February 2011.

[24] V. Narasiman, C.J. Lee, M. Shebanow, R. Miftakhutdinov, O. Mutlu, and Y.N. Patt. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In *44th International Symposium on Microarchitecture (MICRO-44)*, December 2011.

[25] Minsoo Rhu and Mattan Erez. CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures. In *39th International Symposium on Computer Architecture (ISCA-39)*, June 2012.

[26] Minsoo Rhu and Mattan Erez. Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation. In *40th International Symposium on Computer Architecture (ISCA-40)*, June 2013.

[27] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures. In *46th International Symposium on Microarchitecture (MICRO-46)*, December 2013.

[28] NVIDIA Corporation. NVIDIA CUDA Programming Guide, 2011.

[29] AMD Corporation. ATI Stream Computing OpenCL Programming Guide, August 2010.

[30] NVIDIA Corporation. Whitepaper: NVIDIA GeForce GTX 680, 2012.

[31] AMD Corporation. AMD Radeon HD 7900 Series Specification, 2011.

[32] Intel Corporation. Intel HD Graphics OpenSource Programmer Reference Manual, June 2011.

[33] AMD Corporation. Graphics Core Next (GCN) Architecture, 2013.

[34] Mark Gebhart, Daniel Johnson, David Tarjan, Steve Keckler, William Dally, Erik Lindholm, and Kevin Skadron. Energy-Efficient Mechanisms for Managing Thread Context in Throughput Processors. In *38th International Symposium on Computer Architecture (ISCA-38)*, 2011.

[35] Timothy Rogers, Mike O'Connor, and Tor Aamodt. Cache-Conscious Wavefront Scheduling. In *45th International Symposium on Microarchitecture (MICRO-45)*, December 2012.

[36] Adwait Jog and et al. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-13)*, 2013.

[37] Adwait Jog and et al. Orchestrated Scheduling and Prefetching for GPGPUs. In *40th International Symposium on Computer Architecture (ISCA-40)*, 2013.

[38] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[39] AMD Corporation. AMD Radeon HD 6900M Series Specifications, 2010.

[40] Hynix. *1Gb (32Mx32) GDDR5 SGRAM, H5GQ1H24AFR*, 2009.

[41] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[42] AMD Corporation. R700-Family Instruction Set Architecture, February 2011.

[43] Collange, Sylvain. Stack-less SIMT Reconvergence At Low Cost, 2011.

[44] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In *39th International Symposium on Computer Architecture (ISCA-39)*, June 2012.

[45] B. W. Coon, P. C. Mills, S. F. Oberman, and M. Y. Siu. Tracking Register Usage During Multithreaded Processing Using a Scoreboard Having Separate Memory Regions And Storing Sequential Register Size Indicators. In *US Patent 7434032*, October 2008.

[46] GPGPU-Sim. http://www.gpgpu-sim.org.

[47] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA Workloads using a Detailed GPU Simulator. In *IEEE*

International Symposium on Performance Analysis of Systems and Software (ISPASS-2009), April 2009.

[48] GPGPU-Sim Manual. http://www.gpgpu-sim.org/manual.

[49] IMPACT Research Group. The Parboil Benchmark Suite, 2007.

[50] Tianyi D. Han and Tarek Abdelrahman. Reducing Branch Divergence in GPU Programs. In *4-th Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4)*, March 2011.

[51] Abdullah Gharaibeh and Matei Ripeanu. Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance. In *2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC-2010)*, November 2010.

[52] Sunpyo Hong and Hyesoon Kim. An Integrated GPU Power and Performance Model. In *37th International Symposium on Computer Architecture (ISCA-37)*, June 2010.

[53] Jingwen Leng and et al. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *40th International Symposium on Computer Architecture (ISCA-40)*, June 2013.

[54] Glew, A. Coherent Vector Lane Threading. In *Berkeley Parlab Seminar*, 2009.

[55] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili. SIMD Re-Convergence At Thread Frontiers. In *44th International Symposium on Microarchitecture (MICRO-44)*, December 2011.

[56] NVIDIA Corporation. PTX: Parallel Thread Execution ISA Version 2.3, 2011.

[57] Nuwan Jayasena, Mattan Erez, Jung Ho Ahn, and William Dally. Stream Register Files with Indexed Access. In *10th International Symposium on High Performance Computer Architecture (HPCA-10)*, February 2004.

[58] John Hennessy and David Patterson. Computer Architecture: A Quantitative Approach, 4th Edition, 2013.

[59] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Compiler for Bulk-Synchronous Applications in Heterogenous Systems. In *19th International Conference on Parallel Architecture and Compilation Techniques (PACT-19)*, September 2010.

[60] Mike Giles. Jacobi Iteration for a Laplace Discretisation on a 3D Structured Grid, 2008.

[61] Mike Giles and Su Xiaoke. Notes on Using the NVIDIA 8800 GTX Graphics Card. http://people.maths.ox.ac.uk/ gilesm/hpc/, 2008.

[62] Pawan Harish and P. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *High Performance Computing HiPC 2007*, volume 4873, pages 197–208. 2007.

[63] Michael Schatz, Cole Trapnell, Arthur Delcher, and Amitabh Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1):474, 2007.

[64] Samir Palnitkar. Verilog HDL, 2nd Edition, 2003.

[65] Eddy Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. Streamlining GPU Applications On the Fly. In *24th International Conference on Supercomputing (ICS'24)*, June 2010.

[66] Bingsheng He, Wenbin Fang, Qiong Luo, Naga Govindaraju, and Tuyong Wang. A MapReduce Framework on Graphics Processors. In *17th International Conference on Parallel Architecture and Compilation Techniques (PACT-17)*, 2008.

[67] David Tarjan, Jiayuan Meng, and Kevin Skadron. Increasing Memory Miss Tolerance for SIMD Cores. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC-09)*, 2009.

[68] Aniruddha Vaidya, Anahita Shayesteh, Dong Hyuk Woo, Roy Saharoy, and Mani Azimi. SIMD Divergence Optimization through Intra-Warp Compaction. In *40th International Symposium on Computer Architecture (ISCA-40)*, June 2013.

[69] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. In *IEEE Micro*, October 2011.

[70] Intel Corporation. Intel Core i7-4960X Processor Specification, 2013.

[71] R. Kalla, B. Sinharoy, W.J. Starke, and M. Floyd. Power7: IBM's Next-Generation Server Processor, 2010.

[72] J.L. Shin, Dawei Huang, B. Petrick, Changku Hwang, K.W. Tam, A. Smith, Ha Pham, Hongping Li, T. Johnson, F. Schumacher, A.S. Leon, and A. Strong. A 40 nm 16-Core 128-Thread SPARC SoC Processor, 2011.

[73] Doe Hyun Yoon, Min Kyu Jeong, and Mattan Erez. Adaptive Granularity Memory Systems: A Tradeoff between Storage Efficiency and Throughput. In *38th International Symposium on Computer Architecture (ISCA-38)*, 2011.

[74] Doe Hyun Yoon, Michael Sullivan, Min Kyu Jeong, and Mattan Erez. The Dynamic Granularity Memory System. In *39th International Symposium on Computer Architecture (ISCA-39)*, 2012.

[75] Jung Ho Ahn, Norman P. Jouppi, Christos Kozyrakis, Jacob Leverich, and Robert S. Schreiber. Future Scaling of Processor-Memmory Interfaces. In *Proc. the Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2009.

[76] Jung Ho Ahn, Jacob Leverich, Robert Schreiber, and Norman P. Jouppi. Multicore DIMM: An Energy Efficient Memory Module with Indepen-

dently Controlled DRAMs. *IEEE Computer Architecture Letters*, 8(1):5–8, Jan. - Jun. 2009.

[77] F. A. Ware and C. Hampel. Micro-threaded Row and Column Operations in a DRAM Core. In *Proc. the first Workshop on Unique Chips and Systems (UCAS)*, Mar. 2005.

[78] F. A. Ware and C. Hampel. Improving Power and Data Efficiency with Threaded Memory Modules. In *Proceedings of the International Conference on Computer Design (ICCD)*, 2006.

[79] Hongzhong Zheng, Jiang Lin, Zhao Zhang, Eugene Gorbatov, Howard David, and Zhichun Zhu. Mini-Rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency. In *41st International Symposium on Microarchitecture (MICRO-41)*, Nov. 2008.

[80] Tony M. Brewer. Instruction Set Innovations for the Convey HC-1 Computer. *IEEE Micro*, 30(2):70–79, 2010.

[81] Dennis Abts, Abdulla Bataineh, Steve Scott, Greg Faanes, James Schwarzmeier, Eric Lundberg, Mike Byte, and Gerald Schwoerer. The Cray Black Widow: A highly scalable vector multiprocessor. In *Proc. the Int'l Conf. High Performance Computing, Networking, Storage, and Analysis (SC)*, Nov. 2007.

[82] M. McTague and H. David. Fully Buffered DIMM (FB-DIMM) Design Considerations, 2004.

[83] J. S. Liptay. Structural Aspects of the System/360 Model 85, Part II: The Cache. *IBM Systems Journal*, 7:15–21, 1968.

[84] Sanjeev Kumar and Christopher Wilkerson. Exploiting Spatial Locality in Data Caches using Spatial Footprints. In *25th International Symposium on Computer Architecture (ISCA-25)*, 1998.

[85] Chi Chen, Se-Hyun Yang, Babak Falsafi, and Andreas Moshovos. Accurate and Complexity-Effective Spatial Pattern Prediction. In *10th International Symposium on High Performance Computer Architecture (HPCA-10)*, 2004.

[86] Wenhao Jia, Kelly Shaw, and Margaret Martonosi. Characterizing and Improving the Use of Demand-Fetched Caches in GPUs. In *26th International Supercomputing Conference (ICS'26)*, 2012.

[87] Burton Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. In *ACM Communications*, 1970.

[88] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary Cache: a Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[89] Eric Sprangle, Robert S Chappell, Mitch Alsup, and Yale N Patt. The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference. In *17th International Symposium on Computer Architecture (ISCA-17)*, 1997.

[90] M Ramakrishna and et al. Efficient Hardware Hashing Functions for High Performance Computers. In *IEEE Transactions on Computers*, 1997.

[91] DrSim. http://lph.ece.utexas.edu/public/DrSim.

[92] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, Michael Sullivan, Ikhwan Lee, and Mattan Erez. Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems. In *18th International Symposium on High Performance Computer Architecture (HPCA-18)*, February 2012.

[93] Tae-Young Oh and et al. A 7Gb/s/pin 1 Gbit GDDR5 SDRAM With 2.5 ns Bank to Bank Active Time and No Bank Group Restriction. In *IEEE Journal of Solid-State Circuits*, 2011.

[94] Micron Corporation. Calculating Memory System Power for DDR3, 2007.

[95] HMC. Hybrid memory cube specification 1.0, 2013.

[96] Hynix. Blazing A Trail to High Performance Graphics, 2011.

[97] JEDEC. JESD 229 Wide I/O SDR, 2011.

[98] Doe Hyun Yoon and Mattan Erez. Virtualized and Flexible ECC for Main Memory. In *Proc. the 15th Int'l. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2010.

[99] Sheng Li, Doe Hyun Yoon, Ke Chen, Jishen Zhao, Jung Ho Ahn, Jay B Brockman, Yuan Xie, and Norman P Jouppi. MAGE: adaptive granularity and ECC for resilient and power efficient memory systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[100] A. Seznec. Decoupled Sectored Caches: Conciliating Low Tag Implementation Cost. In *Proc. the 21st Ann. Int'l Symp. Computer Architecture (ISCA-21)*, Apr. 1994.

[101] Jeffrey B. Rothman and Alan Jay Smith. The Pool of Subsectors Cache Design. In *Proc. the 13th Int'l Conf. Supercomputing (ICS)*, Jun. 1999.

[102] A. Gonzalez, C. Aliagas, and M. Valero. A Data Cache with Multiple Caching Strategies tuned to Different Types of Locality. In *Proc. the Int'l Conf. Supercomputing (ICS)*, Jul. 1995.

[103] Fan Deng and Davood Rafiei. Approximately Detecting Duplicates for Streaming Data using Stable Bloom Filters. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 25–36. ACM, 2006.

[104] bcache: A Linux kernel block layer cache.

[105] Francis Chang, Wu-chang Feng, and Kang Li. Approximate Caches for Packet Classification. In *INFOCOM 2004. Twenty-third AnnualJoint*

Conference of the IEEE Computer and Communications Societies, volume 4, pages 2196–2207. IEEE, 2004.

[106] C. Ungureanu and et al. TBF: A Memory-Efficient Replacement Policy for Flash-based Caches. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1117–1128, 2013.

[107] MyungKeun Yoon. Aging Bloom Filter with Two Active Buffers for Dynamic Sets. *Knowledge and Data Engineering, IEEE Transactions on*, 22(1):134–138, 2010.

[108] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *13th International Conference on Parallel Architecture and Compilation Techniques (PACT-13)*, 2004.

[109] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *43th International Symposium on Microarchitecture (MICRO-43)*, December 2010.

[110] Onur Kayiran, Adwait Jog, Mahmut Kandemir, and Chita Das. Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs. In *22st International Conference on Parallel Architecture and Compilation Techniques (PACT-22)*, September 2013.

[111] Scott Rixner, William Dally, Ujval Kapasi, Peter Mattson, and John Owens. Memory Access Scheduling. In *27th International Symposium on Computer Architecture (ISCA-27)*, June 2000.

[112] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *8th International Symposium on Computer Architecture (ISCA-8)*, June 1981.

[113] J. Mohd-Yusof, S. Swaminarayan, and T. C. Germann. Co-Design for Molecular Dynamics: An Exascale Proxy Application, 2013.

[114] Jaekyu Lee and Hyesoon Kim. TAP: A TLP-Aware Cache Management Schemes for a CPU-GPU Heterogeneous Architecture. In *18th International Symposium on High Performance Computer Architecture (HPCA-18)*, February 2012.

Vita

Minsoo Rhu studied at Sogang University, Seoul, Korea, where he received a B.E. degree in Electronic Engineering in August, 2007.

Minsoo initially started his graduate study at KAIST (Korea Advanced Instiute of Science and Technology) in 2008, designing high-performance RISC processors and JPEG2000 VLSI encoders. He received a M.S. degree in Electrical Engineering in 2009. He then started his doctoral study in 2010 at the University of Texas at Austin in the areas of computer architecture. In particular, his research is focused on high performance processor architecture and cache/memory hierarchy designs for throughput processors and graphics processing units (GPUs). His research has been published in major computer architecture conferences such as ISCA, MICRO, and HPCA.

E-mail address: minsoo.rhu@gmail.com

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.