The Dissertation Committee for Dong Li
certifies that this is the approved version of the following dissertation:

Orchestrating Thread Scheduling and Cache Management to Improve Memory System Throughput in Throughput Processors

Committee:

Donald S. Fussell, Supervisor

Douglas C. Burger, Co-Supervisor

Stephen W. Keckler

Calvin Lin

Lizy K. John

Orchestrating Thread Scheduling and Cache Management to Improve Memory System Throughput in Throughput Processors

by

Dong Li, B.E.; M.E.

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

To my wife, my children and my parents.

Acknowledgments

I would like to thank many people who have been helping me along my journey to complete this dissertation. First, I would like to thank my advisor, Doug Burger, for his excellent guidance and support during my study. Doug is a great researcher and has been a wonderful mentor. I am grateful to him for dedicating his time to guide my research during these years.

I would also like to thank my advisor, Don Fussell, for his great mentoring and encouragement. Don not only shared his amazing insights into the computer architecture research, but also provided me invaluable career advice. I am very thankful to Steve Keckler. Steve played a significant role in my graduate study. He led me to find interesting research topics in the GPU research area. His great guidance during these years has been a key for me to finish my dissertation.

It has been a big pleasure and my great honor to work closely with Doug, Don and Steve when conducting research on GPU. With the excellent guidance and inspiration from three great technical leaders, I have been enjoying every minute I spent to investigate this research field.

I would like to thank the other members of my committee, Calvin Lin and Lizy John. They have provided helpful feedback along the way. I am also grateful to Kathryn McKinley. Kathryn has provided great technical guidance

during my study. I am very thankful to my colleagues and friends who helped me to complete this work including Jeff Diamond, Behnam Robatmili, Minsoo Rhu, Renee Amant, Curtis Dunham, Sibi Govindan, Akanksha Jain, Ashay Ranee, Jee Ho Ryoo, Hao Wu and Jia Chen. In particular, I'd like to thank Jeff for his deep technical insights and great encouragement during past several years.

Finally, I would like to thank my family. My parents have always been supportive throughout the years. I want to thank my wife, Mei Ming, who has constantly given me support and encouragement. I thank her for being so understanding and for supporting me through the toughest moments of my life. I also dedicate this Ph.D thesis to my two lovely children, Tyler and Kevin. They bring laughs and joy into my life everyday.

DONG LI

The University of Texas at Austin
April 2014

Orchestrating Thread Scheduling and Cache Management to Improve Memory System Throughput in Throughput Processors

Dong Li, Ph.D.
The University of Texas at Austin, 2014

Supervisors: Donald S. Fussell
Douglas C. Burger

Throughput processors such as GPUs continue to provide higher peak arithmetic capability. Designing a high throughput memory system to keep the computational units busy is very challenging. Future throughput processors must continue to exploit data locality and utilize the on-chip and off-chip resources in the memory system more effectively to further improve the memory system throughput. This dissertation advocates orchestrating the thread scheduler with the cache management algorithms to alleviate GPU cache thrashing and pollution, avoid bandwidth saturation and maximize GPU memory system throughput. Based on this principle, this thesis work proposes three mechanisms to improve the cache efficiency and the memory throughput.

This thesis work enhances the thread throttling mechanism with the Priority-based Cache Allocation mechanism (PCAL). By estimating the cache miss ratio with a variable number of cache-feeding threads and monitoring

the usage of key memory system resources, PCAL determines the number of threads to share the cache and the minimum number of threads bypassing the cache that saturate memory system resources. This approach reduces the cache thrashing problem and effectively employs chip resources that would otherwise go unused by a pure thread throttling approach. We observe 67% improvement over the original as-is benchmarks and a 18% improvement over a better-tuned warp-throttling baseline.

This work proposes the AgeLRU and Dynamic-AgeLRU mechanisms to address the inter-thread cache thrashing problem. AgeLRU prioritizes cache blocks based on the scheduling priority of their fetching warp at replacement. Dynamic-AgeLRU selects the AgeLRU algorithm and the LRU algorithm adaptively to avoid degrading the performance of non-thrashing applications. There are three variants of the AgeLRU algorithm: (1) replacement-only, (2) bypassing, and (3) bypassing with traffic optimization. Compared to the LRU algorithm, the above mentioned three variants of the AgeLRU algorithm enable increases in performance of 4%, 8% and 28% respectively across a set of cache-sensitive benchmarks.

This thesis work develops the Reuse-Prediction-based cache Replacement scheme (RPR) for the GPU L1 data cache to address the intra-thread cache pollution problem. By combining the GPU thread scheduling priority together with the fetching Program Counter (PC) to generate a signature as the index of the prediction table, RPR identifies and prioritizes the near-reuse blocks and high-reuse blocks to maximize the cache efficiency. Compared to

the AgeLRU algorithm, the experimental results show that the RPR algorithm results in a throughput improvement of 5% on average for regular applications, and a speedup of 3.2% on average across a set of cache-sensitive benchmarks.

The techniques proposed in this dissertation are able to alleviate the cache thrashing, cache pollution and resource saturation problems effectively. We believe when these techniques are combined, they will synergistically further improve GPU cache efficiency and the overall memory system throughput.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Processors that target throughput computing, such as GPUs and many-core processors, are generally referred to as throughput processors. They have become the dominant architecture for accelerating massively parallel applications as they offer greater instruction throughput and memory bandwidth than conventional CPUs. Such chips are being used to accelerate desktops, workstations, and supercomputers, and throughput computing is now emerging as an important factor for mobile computing.

However, the transistor density of modern processors increases much faster than the off-chip bandwidth does [6] [76]. Throughput processors are expected to process more data in parallel in every new generation. The gap between the peak arithmetic capability and the off-chip bandwidth will inevitably grow. Since memory system throughput has become a bottleneck for GPU performance [40], contemporary GPUs integrate an on-chip cache hierarchy to increase memory system throughput. With a large number of threads sharing the cache hierarchy, GPUs can only supply less than a single cache line on average per thread at L1 level [63] [67]. Such sharing leads to contention and reduces the effectiveness of caches in exploiting locality, reducing

the performance improvement available to GPUs for cache-sensitive workloads. Consequently, exploiting data locality in throughput computing systems will increase in importance to reduce both off-chip bandwidth requirements and power.

In throughput processors, both thread scheduler and cache management logic play important roles in cache efficiency and memory system throughput. The thread scheduler determines the order and time of executing instructions from a potentially large pool of threads. It shapes memory access patterns directly and thus has a significant effect on cache efficiency. Without considering the effect of the thread scheduling algorithm, applying thrashing-resistant and pollution-resistant CPU cache algorithms directly to the GPU cache can not effectively address GPU L1 cache thrashing and pollution problems (We discuss these issues in Chapter 5 and Chapter 6 respectively). Without being aware of cache performance and chip resource usage, the scheduler may oversubscribe threads which leads to cache thrashing and resource saturation problems. **My research goal is to orchestrate thread scheduling and cache management polices, to improve cache efficiency and memory system throughput.**

While our evaluation has simulated NVIDIA GPUs with a CUDA programming model, other throughput processors such as many-core processors [80] [15] are also facing the challenge of improving memory system throughput [50] [25]. Recent many-core processors such as Intel's many-core Larrabee [80] and Many Integrated Core (MIC) [15] processor can support hun-

dreds of hardware threads. The benefits of coordinating the thread scheduling and cache management policies may also apply to these processors.

1.1 Memory Systems in Throughput Processors

GPUs rely on massive multithreading to tolerate memory latency and deliver high throughput. For example, a modern NVIDIA Kepler GPU [67] can have up to 30,000 threads hardware-resident and operating simultaneously. GPU memory systems have grown to include a multi-level on-chip cache hierarchy with both hardware and software controlled cache structures. The top-end NVIDIA Kepler chip now includes a total of nearly a megabyte of primary cache (including both hardware and software controlled cache) and 1.5MB of L2 cache. Many-core processors, which also target throughput computing instead of single-thread performance, employ limited per-core threading and relatively larger data caches. The difference between GPUs and many-core memory systems is highlighted in Table 1.1, which summarizes the thread and cache capacity of contemporary throughput processors. Prior research demonstrates that the cache hierarchy in many-core processors is still essential for good performance and energy efficiency [25]. The primary cache capacity per thread for GPUs is 2 to 3 orders of magnitude smaller than for a many-core processor. GPUs can only supply less than a single cache line of space on average, per thread. As a result, keeping the working set of all threads resident in the primary cache is infeasible on a fully occupied streaming multiprocessor (SM);

Table 1.1: Per-thread L1 cache capacity of modern throughput processors

	NVIDIA Kepler [67]	**NVIDIA Fermi** [63]	**Intel Larrabee** [80]	**Intel MIC** [15]	**Oracle T3** [83]
L1 Size	48 KB L1	48 KB L1	32 KB L1	32 KB L1	8 KB L1
Threads /Core	2,048	1,536	4	4	8
L1 Capacity /thread	24 B	32 B	8,192 B	8,192 B	1,024 B

1.2 Key Factors that Affect GPU Memory Throughput

Memory systems have been well-known performance bottlenecks for both CPUs and throughput processors [6] [26] [76] [40] [25] [54]. For throughput processors, there are three major factors that affect the cache efficiency and the memory system throughput dramatically, namely cache management policies, GPU thread scheduling and on-chip/off-chip resource saturation.

1.2.1 Cache Management Policies

Compared to a CPU cache, a GPU cache exhibits two significant differences. First, as we demonstrate in Section 1.1, the limited per-thread cache capacity causes a significant cache thrashing problem. Second, the cache access stream is a mix of requests from many threads. The reuse pattern of a cache block is often largely affected by the scheduling priority of the threads fetching the block.

Cache management policies, such as replacement, bypassing and allo-

cation algorithms etc., have a direct effect on cache efficiency and thus affect memory system throughput significantly. Parts of this dissertation focus on enhancing cache replacement, bypassing and allocation algorithms by coordinating with the GPU thread scheduling to improve cache efficiency.

An optimal cache replacement algorithm always evicts the cache blocks that will be referenced furthest in the future. Unfortunately, oracular knowledge about the future reuse of cache blocks is not available in reality, thus cache replacement algorithms predict the reuse interval of all blocks based on past references. The most common replacement algorithms, such as Least Recently Used (LRU) and its approximation Not Recently Used (NRU) [27] [87], are often based on an assumption that cache access patterns are recency-friendly. However throughput processors' cache access patterns are often highly affected by thread interleaving. The reuse pattern of a cache block could be determined by the thread scheduling algorithm which may radically change reuse distances. For instance, the Greedy-Then-Oldest (GTO) thread scheduling algorithm prioritizes the warps based on their fetch order (i.e. age). A reuse in a young warp is likely to be interrupted by requests from the older warps. The mismatch between the replacement algorithm and cache access patterns is the root cause that leads to GPU cache inefficiency.

Cache thrashing occurs when the working set of an application is too large to fit in the cache. It is one of the major problems that degrade GPU cache efficiency. GPUs support tens of thousands of hardware threads and hide long-latency operations by issuing instructions from unstalled threads.

Each of the GPU threads normally processes a portion of the input data. The total working set size of a GPU program often is proportional to the number of active threads. The limited GPU cache capacity often cannot capture the total working set of all threads resulting in frequent cache eviction and thus low cache efficiency.

Cache pollution is defined as the problem that occurs when cache blocks with little reuse or no reuse evict the high-reuse blocks. The per-thread cache capacity of the GPU is very limited. The low-reuse blocks waste the limited cache capacity and degrade cache efficiency. It is important that the no/low-reuse cache blocks can be identified so that the replacement algorithm can evict them earlier than the others.

The thrashing and pollution problems often manifest themselves at the same time [68] [91]. This requires complex replacement algorithms to learn the reuse pattern of cache blocks and to achieve higher cache efficiency. In this dissertation, we propose new replacement and bypassing algorithms to improve the cache efficiency. For each block, the new algorithms consider the scheduling priority of the fetching thread and the reuse behavior of the block to improve the replacement and bypassing algorithms.

1.2.2 GPU Thread Scheduling

Unlike CPUs that only support a small number of hardware thread contexts, throughput processors support a large number of hardware thread contexts to hide the memory access latency by fast context-switching among

these threads. The GPU hardware thread scheduler selects warps which already have all the operands ready for execution. The scheduling algorithm has a significant effect on cache access patterns. Simple round-robin scheduling algorithms check the readiness of all warps one-by-one. Each warp gets roughly equal time-slice. Although loose round robin policies promote fairness, GPUs, unlike CPUs, are designed to provide high throughput. Inter-thread fairness is not the design goal of a GPU. Furthermore, round robin policies allow all warps to be active hence lead to a large aggregated working set, which can result in GPU cache thrashing. Prior research proposes the Greedy then Oldest (GTO) algorithm, which favors the oldest warp and minimizes the total working set. It has been proven [77] that the GTO algorithm provides the highest throughput on average compared to other common thread scheduling policies on cache-sensitive workloads.

We observe that the cache replacement policy and the thread scheduling policy can coordinate to increase the cache efficiency and thus improve the total throughput. We enhance the GPU thread scheduler with a cache allocation mechanism, which not only decides the issue order of warps but also decides the cache allocation policy for the memory access in each warp. Our results show the new scheduler enables GPUs to achieve significant speedup.

1.2.3 Chip Resource Saturation

If it does not hit in the L1 or L2 cache, a memory request needs a collection of chip resources to fetch a data block from the main memory. In this

dissertation, we refer to the collection of needed resources to service memory requests as chip resources, which include L1 cache blocks, L1 Miss Status Holding Registers (MSHR) table entries, L2 cache blocks, L2 MSHR entries, DRAM controller scheduling queue entries, Network-on-Chip (NoC) bandwidth, and the off-chip bandwidth. A reservation failure of any of these resources stalls the request and limits the overall memory system throughput.

The off-chip bandwidth has not grown as fast as the transistor density. Consequently, off-chip bandwidth is often a major performance bottleneck. On-chip resources, such as the Network-on-Chip bandwidth and the MSHR table, are designed to satisfy the resource requirements of the common case not the worst case. However, when a GPU program presents a high memory divergence, a warp instruction requests access to multiple cache lines resulting in multiple cache misses. The memory request stream thus becomes bursty and is likely to saturate the on-chip and off-chip memory bandwidth in a short period. The queuing latency increases dramatically when a resource reaches its saturation point. Furthermore, the thread scheduler issues more instructions from other warps to hide the latency, which not only further increases the memory access latency, but also increases the aggregated working set size. As a result, both the off-chip and on-chip bandwidth saturation can degrade memory system throughput.

In our research, the scheduling and replacement policies are further optimized to reduce NoC traffic by fetching only a portion of a cache line for the memory requests that are characterized as bypassing requests. Experi-

mental results show that this is key to improving overall throughput for some benchmarks.

1.3 Dissertation Contribution

This dissertation advocates orchestrating the thread scheduler with the cache management algorithms for massively multi-threaded throughput processors to alleviate GPU cache thrashing, avoid bandwidth saturation and maximize GPU memory system throughput. Three independent mechanisms have been proposed based on this principle. As future work, we expect the three techniques could be combined to allow them work synergistically to further improve the throughput. The major contributions of this work are as follows.

1. **Investigating the memory system effect of high TLP and identifying the performance bottleneck of each application**

   In order to investigate how parallelism affects GPU performance, the cache hit ratio and other chip resource utilization, we characterize a set of cache-sensitive applications by varying the maximum number of warps that each thread scheduler allows. We analyze the Instruction Per Cycle (IPC), L1/L2 cache miss ratio and the chip resource utilization metrics including the memory request round trip latency, the Network-on-Chip (NoC) transmission latency and the DRAM bandwidth utilization. We identify the major performance bottlenecks for each application when

TLP is higher. Applications are grouped into categories based on the bottlenecks.

2. **Addressing cache thrashing and memory system resource saturation with a thread-scheduling directed cache allocation mechanism**

   We enhance the thread throttling technique with the Priority-based Cache Allocation (PCAL) mechanism to address inter-thread L1 cache thrashing and memory system resource saturation. Unlike thread throttling approaches which force all threads to feed the L1 cache, PCAL explicitly determines the number of threads to allocate and fill the cache and the minimum number of threads bypassing the cache to saturate memory system resources. PCAL can improve performance with two optimization strategies: either increasing TLP while maintaining cache hit ratio, or optimizing cache hit ratio while maintaining TLP. This approach reduces the cache thrashing problem and effectively employs the chip resources that would otherwise go unused by a pure thread throttling approach. We observe 67% improvement on average over the original as-is code and a 18% improvement on average over a better-tuned warp-throttling baseline.

3. **Developing the AgeLRU and Dynamic-AgeLRU GPU-specific thrashing-resistant cache replacement and bypassing algorithms**

   We observe that applying thrashing-resistant CPU cache algorithms,

such as Bimodal Insertion Policy (BIP) [71] and Dynamic Insertion Policy (DIP) [71] to the GPU cache can not effectively address GPU L1 cache thrashing. The primary reasons are: (1) BIP randomly selects memory blocks to reside in the cache. It tends to activate all warps concurrently and thus increase the size of total working set. (2) DIP relies on the set-dueling mechanism to evaluate two algorithms on different sets of the same cache. In a GPU, the order of thread interleaving and cache access patterns may change when set-dueling is applied to estimate two algorithms.

We propose the AgeLRU and Dynamic-AgeLRU mechanisms, which are thread scheduling-aware replacement and bypassing algorithms to overcome the thrashing problem. When selecting a collection of memory blocks to reside in the cache, AgeLRU minimizes the number of warps that fetch the cache-resident blocks by prioritizing older warps at replacement. Dynamic-AgeLRU selects the AgeLRU or the LRU algorithm adaptively, based on the parallel voting mechanism.

There are three variants of the AgeLRU algorithm: (1) replacement-only, (2) bypassing, and (3) bypassing with traffic optimization. Compared to the LRU algorithm, the above mentioned three variants of the AgeLRU algorithm enable increases in performance of 4%, 8% and 28% respectively across a set of cache-sensitive benchmarks.

Our results show that Dynamic-AgeLRU algorithms can avoid degrading the performance of non-thrashing applications by selecting the LRU

algorithm.

4. **Proposing Reuse-Prediction based Cache Replacement Algorithm**

   We observe that CPU cache pollution-resistant algorithms can not be applied directly to a GPU L1 cache. The reason is that the GPU thread scheduler has a significant effect on the GPU cache access pattern. Without considering the effect of the thread scheduling, a high accuracy reuse prediction is not achievable.

   We propose a Reuse-Prediction-based cache Replacement (RPR) scheme for a GPU L1 data cache to address the intra-thread cache pollution problem. To increase the prediction accuracy, RPR uses the GPU thread scheduling priority to generate a signature which is the index of the prediction table. RPR can approximate the reuse-distance-based algorithm, the counter-based algorithm, and the AgeLRU algorithm with various cache block score functions. Among these three configurations, our results show that the reuse-distance-based algorithm outperforms the others. Compared to the AgeLRU algorithm we propose in Section 5, the reuse-distance-based algorithm enables a throughput improvement of 5% on average for regular applications, and a speedup of 3.2% across a set of cache-sensitive benchmarks.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows: Chapter 2 discusses the related work. Chapter 3 introduces the background of our research, and describes the baseline GPU architecture and the CUDA programming model. It also contains characterizations of CUDA applications. The cache sensitivity and the scheduling algorithm sensitivity are investigated. Our experimental methodology is also discussed. Chapter 4 describes the PCAL scheme, which is the thread scheduler directed cache allocation mechanism to address the inter-thread L1 cache contention and the chip resource saturation problem. Chapter 5 proposes AgeLRU, a thread-scheduling-aware cache replacement and bypassing scheme to address the inter-thread cache contention problem. Chapter 6 introduces RPR, a reuse-prediction based cache replacement policy to alleviate intra-thread cache pollution. Chapter 7 draws conclusions from this work and discusses directions for future study.

Chapter 2

Related Work

This section summarizes prior parallelism tuning and cache management work for both GPUs and CPUs.

2.1 Tuning Parallelism to Improve Performance

2.1.1 Parallelism Tuning for GPUs

Bakhoda, et al. [4] evaluate a number of workloads across several GPU configurations in which the maximum number of supported CTAs varies. They note that some workloads perform better when the number of concurrent CTAs is limited, and attribute this effect to reduced contention for shared resources (e.g. caches, interconnection network, and DRAM bandwidth). Kayiran, et al. [39] propose a mechanism to dynamically determine the best number of CTAs to concurrently schedule to attain the highest performance in light of these memory system contention effects. They monitor whether a CTA is experiencing a high number of memory-induced stalls or whether it is unable to keep the machine busy, and dynamically adjust the number of CTAs allowed to execute on an streaming multiprocessor (SM) to achieve a balance, leading to good performance. Rogers, et al. [77] [78]directly address L1 data cache

contention among threads. Their cache-conscious wavefront scheduling technique monitors the L1 cache using a victim tag cache to detect when locality is being lost due to cache contention among warps. As lost-locality is detected, certain warps are made ineligible to issue additional memory requests, thereby preserving locality for the remaining warps. Several researchers have also proposed a variety of schedulers that preferentially schedule out of a small pool of warps [16, 59, 18, 37]. These two-level schedulers have been developed for a number of reasons, but all of them generally have the effect of reducing contention in the caches and memory subsystem by limiting the number of co-scheduled warps. Guz et al. [22] [21] describe "the performance valley" that exists between highly-threaded systems where threading can hide the memory latency and systems in which the working set of the active threads fits primarily within the cache.

These thread scheduling policies generally improve the performance for cache-sensitive applications by reducing thread-level parallelism to avoid entering this performance valley. Our approach does not reduce available thread-level parallelism by throttling threads, but rather seeks to directly reduce the memory system contention effects by providing preferential service to only a subset of warps (e.g. those likely to be enabled in one of the thread scheduling policies above). At the same time, our approach allows the remaining threads to remain active, opportunistically takes advantage of inter-warp locality with the warps holding tokens, and exploits available "spare" bandwidth through the memory system. In effect, our approach seeks to eliminate the "perfor-

mance valley" - allowing high thread-level parallelism while mitigating the memory system contention issues that affect these highly-threaded systems.

2.1.2 Parallelism Tuning for CPUs

Nieplocha, et al. [62] note that having too many threads may hurt the performance for some scientific applications. Suleman, et al. [86] propose the first dynamic thread throttling technique for Chip Multiprocessors (CMP). A feedback driven threading (FDT) technique has been proposed to dynamically decide the best number of threads of CMPs. The technique attempts to avoid the synchronisation overhead of executing critical sections sequentially from all threads. It also targets finding the minimum number of threads to saturate the off-chip bandwidth. Compared to our work, this work is a pure thread throttling technique, and it does not attempt to avoid inter-thread cache contention problem.

Along with [86], several other dynamic parallelism management approaches have been proposed [84] [53] [61] [49] [101]. Among these techniques, Sridharan, et al. [84]describe a new technique that tunes the degree of parallelism dynamically adapting to execution condition changes. Luo, et al. [53] propose to accelerate single thread applications by dynamic spawning speculative threads at runtime. Nicolau, et al. [61]propose a technique to decide when and how many threads should be spawned at runtime. Lee, et al. [49] describe a dynamic thread throttling and merging technique which targets optimize both cache efficiency and inter-thread communication overhead.

Compared to our work, all these studies [86] [84] [53] [61] [49] [101] [12], are thread throttling techniques, the only parameter to tune the parallelism is the total number of threads. Our PCAL work separates the concerns of saturating different resources, the total number of threads does not have to be the number for threads to feed the L1 cache or the L2 cache.

2.2 GPU Cache Allocation and Replacement Policies

Jia et al. [33] describe a compile-time algorithm to determine whether to selectively enable cache allocation in the L1 data cache for each load instruction in the program. The algorithm does not attempt to predict temporal cache reuse, but rather focuses on the expected degree of spatial locality among the accesses within a warp. Accesses that are anticipated to require many cache lines to satisfy are marked to bypass the L1 data cache. In contrast to our work, this work identifies the static loads that have the most potential to require a large number cache evictions, and prevents these loads from disrupting the cache. Our work focuses on adaptively determining a subset of the warps that should be enabled to cause cache allocations - allowing temporal reuse, even for loads performing highly divergent accesses. In addition, their compile time analysis, unlike our dynamic system, is unable to handle situations in which the locality is input data dependent.

Jia, et al. [34] propose two mechanisms, request-reordering and bypassing-at-stall, to improve GPU cache efficiency. The request reordering mechanism, applies per-warp queues to hold and group memory requests from same warp.

It prevents intra-warp reuse pairs from being interrupted by requests from other warps if the reuse pairs both stay in the queue before accessing cache. The bypassing at stall mechanism allows a memory request to bypass the cache when all cache blocks in a cache set have been reserved. Unlike our work, the bypassing at stall policy only considers the chip resource congestion problem. When chip resources are not congested, memory requests from all warps still can feed the L1 cache and lead to the cache thrashing problem. This policy can be considered as a special case of the bypassing policy we propose for the Priority based Cache Allocation (PCAL) mechanism.

Rhu, et al. [75] proposes an energy-efficient GPU memory hierarchy, which predicts spatial reuse patterns of cache blocks and only fetches a portion of each cache block to save energy consumption and reduce memory bandwidth requirement. Our PCAL work does not predict spatial locality. PCAL only allows bypassing loads to not fetch whole cache blocks.

GPU prefetching techniques [48] [36] [82] have been proposed to utilize off-chip bandwidth more effectively and reduce memory access latency to increase GPU occupancy. The mechanisms this dissertation proposes do not consider prefetching. However, both AgeLRU and RPR are able to improve cache efficiency and thus reduce off-chip bandwidth requirement. Prefetching might complement our techniques to utilize the off-chip bandwidth.

GPU cache analytical models [69] [89] [8] have been proposed based on stack distance [56] and reuse distance [5]. These studies demonstrate that reuse distance and stack distance are fundamental to model GPU cache behavior.

This dissertation proposes RPR mechanism to predict reuse distance of GPU cache blocks dynamically to improve GPU cache replacement decision.

Software and compiler techniques [102] [11] [88] [94] [100] [93] [99] have been proposed to improve data layout so that memory system throughput can be improved. These studies generally focus on software techniques to optimize data locality. This dissertation proposes techniques to optimize thread scheduler and cache management which do not change the source code. We expect our techniques and these software techniques might work synergistically to further improve GPU cache efficiency and the overall memory system throughput.

2.3 CPU Cache Allocation and Replacement Policies

Cache thrashing, the cache pollution and dead block problem are common issues that degrade cache efficiency. Many thrashing-resistant [51] [71] [81] [30], pollution-resistant [47] [31] [97] [42] [81] [13] [55] [14] [35] and dead block prediction [79] [96] [45] [46] [43] [52] techniques have been proposed to address these problems. We summarize the most relevant CPU cache studies and compare our work with them.

Lin, et al. [51] propose to assign low replacement priorities to prefetches. The prefetched blocks are placed in non-MRU point of the replacement priority chain thus they do not thrash the demand-miss blocks. Qureshi, et al. [71] propose several thrashing-resistant cache adaptive insertion algorithms. Among them, Bimodal Insertion Policy (BIP) randomly selects a small fraction of all

cache blocks to reside in the Most Recently Used (MRU) state while the majority of cache blocks are kept in the LRU state. BIP allows a fraction of working set to reside in cache when the programming working set is much bigger than the cache capacity. However, BIP hurts the performance for the no-thrashing workload. The paper proposes the Set-Dueling mechanism which allows multiple replacement/insertion policies to be compared to select one at runtime. Applying the Set-Dueling mechanism beyond BIP, a Dynamic Insertion Policy (DIP) is proposed to choose between BIP and LRU adaptively. Compared to BIP/DIP, the AgeLRU replacement policy we propose targets the thrashing problem of GPU caches. AgeLRU considers the thread scheduler's effect on the memory access patterns. While BIP/DIP randomly selects cache blocks to stay at MRU position, AgeLRU assigns higher priority to the cache blocks fetched by the older warps.

Lee, et al. [47] propose the Least Frequently Used replacement (LRU) algorithm to avoid no reuse memory requests from polluting the cache. It is able to identify the no reuse or low-reuse cache blocks based on the reuse frequency but cannot capture the reuse from the recency-friendly access patterns.

Jaleel, et al. [31] propose a replacement policy named Static Re-reference Interval Prediction (SRRIP) to address the cache pollution problem, and extend SRRIP to Dynamic Re-reference Interval Prediction (DRRIP) to address the cache thrashing problem. Both of the algorithms are implemented based on the Not Recently Used (LRU) [27] [87] replacement algorithm. SRRIP predicts that cache blocks that get reused after being fetched are likely to be

reused again. SRRIP promotes cache blocks on cache hit and partially solves t he cache pollution problem. This paper also proposes DRRIP, which applies the Set-Dueling mechanism to select SRRIP and LRU dynamically. However, SRRIP can perform the promotion only if there is reuse happening. When the length of the scan pattern is longer than the cache capacity, SRRIP does not have an opportunity to promote reusable cache blocks and thus cannot filter out the no-reuse blocks. Therefore, SRRIP and DRRIP only partially solve the cache pollution problem. The root reason that SRRIP cannot fully solve the cache pollution problem is that it can only learn the reuse behavior of a cache block when the block is resident in cache. Even if SRRIP learns the reuse pattern of a cache block, it cannot store it externally, thus the learned pattern cannot be applied to this block again or to the other cache blocks that may share similar reuse behavior.

Wu, et al. [97] propose the Signature based Hit Predictor (SHiP) to improve on SRRIP [31]. Unlike SRRIP, SHiP does store the learned reuse pattern of cache blocks. but rather stores the hit/miss history of cache blocks and associates the history with a corresponding signature. The memory references that share the same signature are expected to share the reuse pattern. SHiP categorizes memory references into different groups based on the signature of selected attributes of each memory reference. For example, the signature can be formatted by hashing the Program Counter (PC), memory region address or instruction sequence history. As a result, SHiP can predict whether a cache block gets reused or not based on its signature and the corresponding hit/miss

history of this signature. It can place a cache block in the MRU or LRU state based on the predicted hit/miss, thus be able to avoid polluting the cache with no-reuse block .

Keramidas, et al. [42] propose a reuse-distance prediction based cache replacement policy. Compared with SHiP [97], it not only stores and applies the reuse hit/miss history of cache blocks but also the reuse-distance history of cache blocks. With the additional information, this new technique can not only predict whether a cache block hits but also when the reuse happens.

Compared to these thrashing-resistant or pollution-resistant CPU cache management techniques, our work, AgeLRU and RPR, not only leverage the insights from these techniques to address similar problems for GPU caches, but also consider GPU specific attributes to better address GPU cache problems. For example, unlike SHiP [97] which categorizes memory references based on PC or memory region address, our approaches also apply the age of a warp as a factor during categorization.

Dead block problem is also one of the major problems degrading cache efficiency. There are five types of dead block predictor that have been proposed: (1) software based dead block identifying mechanism [79] [96], (2) instruction sequence based predictor [45], (3) timing based predictor [46], (4) counter based predictor [43], and (5) burst based predictor [52]. Compared to these techniques, our work AgeLRU algorithm considers the effect of GPU thread scheduling and utilizes the fetching warps ID to evict the dead cache blocks that are fetched by inactive warps. Unlike th CPU cache dead block

predictors, AgeLRU does not need extra prediction table to store reuse information.

Cache bypassing has been widely exploited in CPU to avoid pollution [19] [32] [38] [92]. These studies rely on cache controllers to make the bypassing decision. In contrast, our work PCAL relies on the scheduler to choose cache threads and bypassing threads.

Other research has been done on partitioning CMPs caches [70] [98] [9] [24] [28] [103] [58] [60] [41]. This work generally deals with multi-programmed workloads, often focusing on fairness and other quality-of-service metrics to ensure one thread does not use a disproportionate share of cache/memory system resources starving the remaining threads. In contrast, our work targets GPU caches. Inter-thread fairness is not the design target of such throughput processors.

Chapter 3

Background

In this chapter, we provide the necessary background for the reader. We begin with contemporary GPU architecture and the CUDA programming model. Next, we characterize a collection of CUDA applications. We investigate their cache sensitivity and select the subset of applications that are cache-sensitive to evaluate our work. We also study the sensitivity of the benchmarks to the thread scheduling policies, and demonstrate that the Greedy-Then-Oldest (GTO) algorithm performs better than other common scheduling algorithms. Finally, our experimental methodology is detailed.

3.1 Contemporary GPU Architecture

Using NVIDIA terminology, modern GPUs consist of many streaming multiprocessor cores (SMs) on a chip, along with a logically shared but physically banked on-chip L2 cache and multiple high-bandwidth memory controllers. For example, our baseline throughput processor, NVIDIA's GTX480 Fermi GPU, has 15 SMs, 768KB of on-chip L2 cache, and 6 GDDR5 memory controllers [63]. Figure 3.1 shows the chip level overview of such a GPU. On this chip, each SM includes a 32KB register file, many parallel arithmetic

pipelines, a 64KB local SRAM that can be split between an L1 cache and a software controlled scratchpad memory, and the capacity to execute up to 1,536 threads. Each SM can support multiple warp schedulers, and each warp scheduler selects instructions from a subset of warps. Figure 3.2 provides a detailed diagram of a SM. In this model, the L1 cache that can be carved out of the local SRAM is 16KB or 48KB.

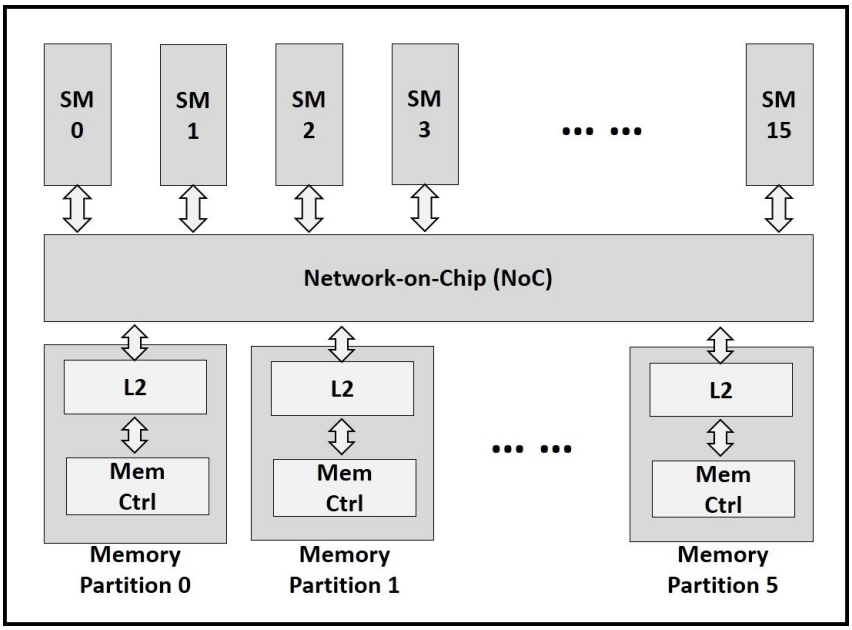


Figure 3.1: Chip level overview of baseline GPU

3.2 CUDA Programming Model

In the CUDA programming model [65], a parallel program is decomposed into kernels or grids that consist of multiple thread blocks or cooperative thread arrays (CTAs). In contemporary GPUs [63] [67], a CTA contains up

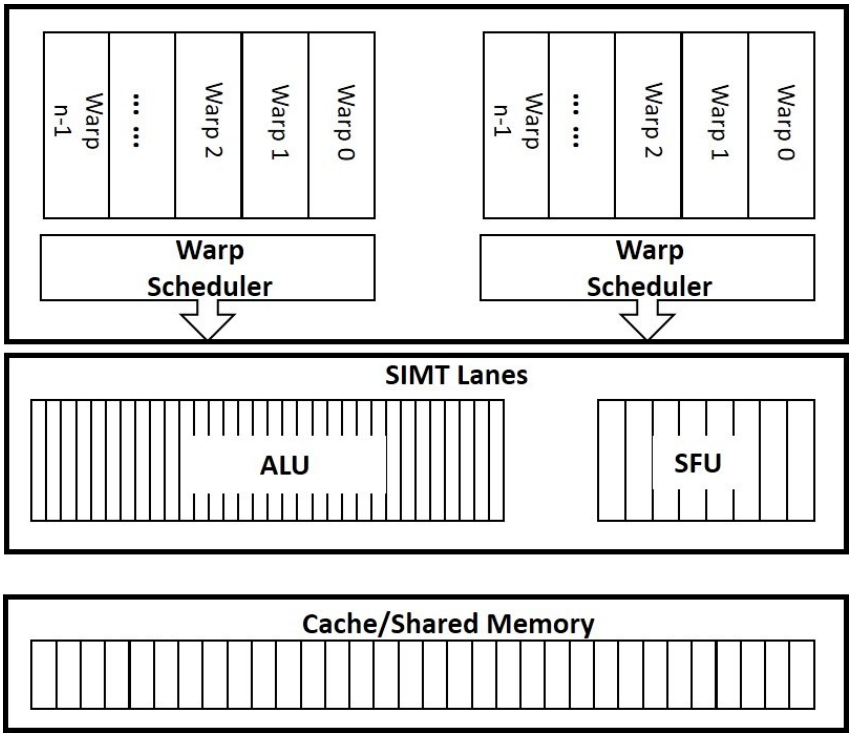


Figure 3.2: Streaming multiprocessor architecture.

to 1024 threads. The execution of CTAs can be performed in any order. Programmers are encouraged to expose as much parallelism as possible as the GPU driver and hardware handles the mapping and scheduling of parallel tasks to the SMs. Each CTA consists of one or more warps, each of which has 32 threads that execute together in a SIMD fashion.

3.3 Thread Scheduling

When a GPU launches a CUDA application, it enqueues the kernels into a streaming queue and dispatches them in order. In our work, GPUs execute kernels sequentially. A global CTA scheduler [65] dispatches CTAs to all SMs until the SMs can not support more CTAs. The maximum number of CTAs each SM can launch simultaneously is determined by the resource constrains on each SM including the maximum number of threads, the share memory size and the register file size. The total size of the share memory space that the launched CTAs declare can not exceed the share memory capacity on each SM. Similarly, the launched CTAs can not require a register file size that is larger than the register file capacity on each SM. After a SM launches CTAs, each warp scheduler on the SM selects and issues instructions from a subset of warps. In this work, we study the interaction between warp scheduling algorithms and cache management algorithms. We often refer the warp scheduler to the thread scheduler in this dissertation.

The most commonly used thread scheduling algorithms are Loose-Round-Robin (LRR), Two-Level (TwoLev) and Greedy-Then-Oldest (GTO). Each of

the algorithms is summarized briefly below:

- Loose-Round-Robin (LRR)

  The LRR algorithm checks whether a warp is ready to execute in a round-robin order, hence that each warp roughly has an equal opportunity to execute. The LRR algorithm tends to activate all warps on each SM. All warps actively access their data working sets, often thrashing the L1D cache and degrading the overall throughput.

- Greedy-Then-Oldest (GTO)

  The GTO algorithm prioritizes a single warp until it stalls, then it prioritizes the warps based on their fetch order. The GTO algorithm minimizes the total number of active threads (i.e. the thread that has outstanding memory requests), and the oldest warp tends to have higher priority to execute.

- Two-Level (TwoLev)

  The thread scheduler divides warps into fetch groups. During scheduling, the scheduler prioritizes the warps in current fetch group. It does not check the warps in the next fetch group until all warps in the current fetch group stall.

3.4 Application Characterization

In this section, we describe the benchmarks we evaluate, introduce our criteria to select cache-sensitive benchmarks, and analyze the effect that the

thread scheduling algorithms have on cache hit ratio and overall throughput.

3.4.1 Cache-Sensitive Benchmarks

Benchmark Selection Criteria: To evaluate our work, we select cache-sensitive benchmarks from a large collection of applications that include NVIDIA SDK [64], PolyBench [20], Parboil [85], Mars [23], Rodinia [10], LonestarGPU [7] and CoMD [29]. Due to long simulation periods, we execute some applications only up to the point where overall IPC exhibits small variations among different iterations of the kernel. We select cache-sensitive benchmarks using the following criteria: all the benchmarks are simulated with the baseline configuration as in Table 3.4 plus a large-cache configuration where the L1 and L2 capacity are increased to 16 times larger of the baseline configuration. An application is classified as cache-sensitive if the large-cache configuration achieves a speedup of 2X or higher compared to the baseline configuration. For applications from different benchmark sets but with similar functionality, only one of multiple instances is kept. Table 3.1 shows the speedup that the large-cache configuration achieves over the baseline configuration. Table 3.2 lists the selected cache-sensitive benchmarks based on the benchmark selection criteria.

3.4.2 Thread Scheduling Effects on Cache Efficiency and Throughput

GPU cache access patterns are largely shaped by the thread scheduler. The scheduling algorithm affects the cache efficiency significantly. Cache

Table 3.1: Speedup normalized to the baseline Fermi configuration, when increasing L1 and L2 cache capacity to 16X

Benchmark	**Speedup**	**Benchmark**	**Speedup**
2DCONV	1.00	MM	1.11
3DCONV	1.39	MONTECARLO	1.13
3MM	1.07	MRI-GRIDDING	1.01
ATAX	2.99	MRI-Q	1.00
BACKPROP	1.10	MST	1.52
BFS	2.30	MUM	1.47
BH	1.08	MUMMERGPU	1.48
BICG	2.70	MVT	2.90
B+TREE	1.10	MYOCYTE	1
CFD	2.23	NN	1.02
COMD	2.45	NQU	1
CORR	1.01	NW	1.14
COVAR	1.00	PARTICLEFILTER	1.14
CUTCP	1.00	PVC	1.98
FASTWT	0.97	PVR	1.91
FDTD-2D	1.01	RAY	1.03
GAUSSIAN	1	SAD	0.99
GEMM	1.10	SCALARPROD	1.00
GESUMMV	3.36	SGEMM	1.07
GRAMSCHM	1.15	SM	1.63
HEARTWALL	1.03	SP	2.05
HISTO	1.37	SPMV	1.76
HOTSPOT	0.99	SRAD_V1	1.44
IBFS	3.38	SRAD_V2	1.01
II	5.05	SS	2.38
KMEANS	1.59	SSSP	2.20
KMN	7.46	STENCIL	1.05
LBM	1.08	STO	1
LEUKOCYTE	0.99	SCLUSTER	6.14
LIB	1.77	SYRK_D	5.28
LPS	1.01	TPACF	1.00
LS_BFS	2.09	WC	2.73
LUD	1.01	AVG	1.78

Table 3.2: Cache-sensitive CUDA benchmarks

Abbreviation	**Description**	**Ref.**
CoMD	Molecular Dynamics	[29]
II	Inverted Index	[23]
BFS	Breadth first search	[7]
CFD	CFD solver	[10]
KMN	K-means	[77]
SM	String Match	[23]
SS	Similarity Score	[23]
SCLUSTER	Streamcluster	[10]
SSSP	Single-source shortest paths	[7]
ATAX	Matrix-transpose-vector multiply	[20]
BICG	BiCGStab linear solver sub-kernel	[20]
GESUMMV	Scalar-vector-matrix multiply	[20]
MVT	Matrix-vector-product transpose	[20]
SP	Survey Propagation, a heuristic SAT-solver	[7]
WC	Word Count	[23]

Table 3.3: Classifying Benchmarks by scheduler preference

Groups	**Definition**	**Number of Benchmarks**
LRR-Preferred	LRR algorithm outperforms GTO by more than 5%	5
GTO-Preferred	GTO algorithm outperforms LRR by more than 5%	18
No Preference	The throughput difference between LRR and GTO is less than 5%	42

oblivious replacement algorithms including LRU are not able to capture the scheduler-algorithm specific reuse patterns. Prior research [77] demonstrates that the GTO algorithm performs best among these scheduling algorithms. We evaluate the LRR, GTO and TwoLev scheduling algorithms across 65 benchmarks.

Figure 3.3 shows the speedup of the GTO algorithm normalized to the LRR algorithm. A benchmark with a speedup value smaller than one means the benchmark works better with the LRR algorithm. Likewise, a speedup value bigger than one means the corresponding benchmark works better with the GTO algorithm. To better understand the thread scheduling algorithm effects on performance, we classify all benchmarks into three categories by their scheduling algorithm preference. The results are shown in Table 3.3. Among the 65 benchmarks we evaluate, 42 benchmarks are not sensitive to the scheduling algorithm. 18 benchmarks achieve more than 5% speedup with the GTO algorithm while there are only 5 benchmarks that perform better with the LRR algorithm. Figure 3.4 compares the L1 data cache miss rate

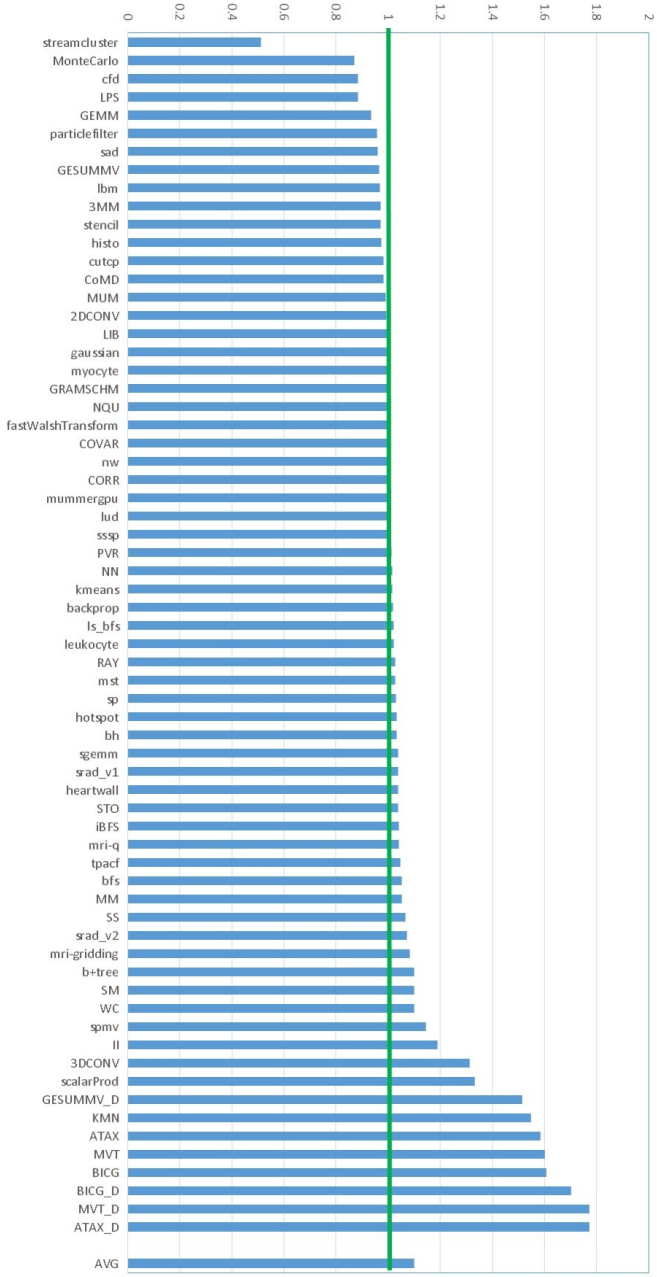


Figure 3.3: Speedup of Greedy-Then-Oldest (GTO) algorithm, normalized to throughput of Loose-Round-Robin (LRR) Scheduling algorithm

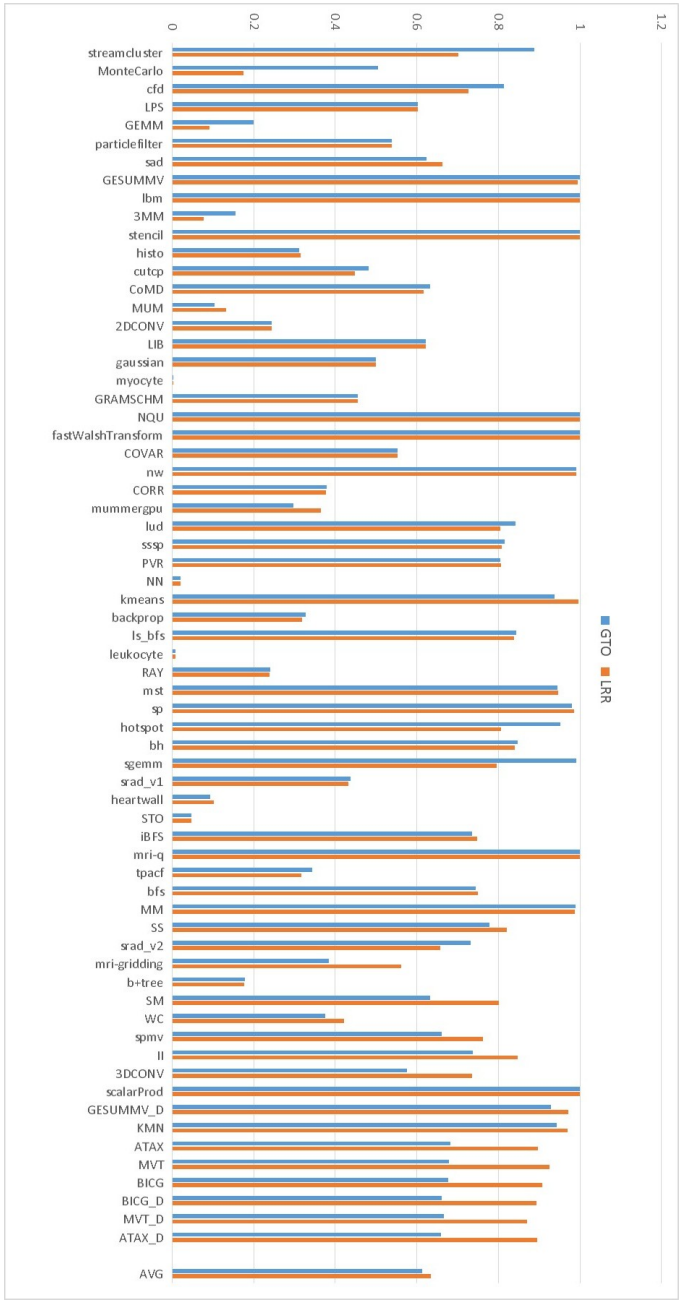


Figure 3.4: Comparing L1 data cache miss rate of Greedy-Then-Oldest (GTO) algorithm and Loose-Round-Robin (LRR) scheduling algorithm

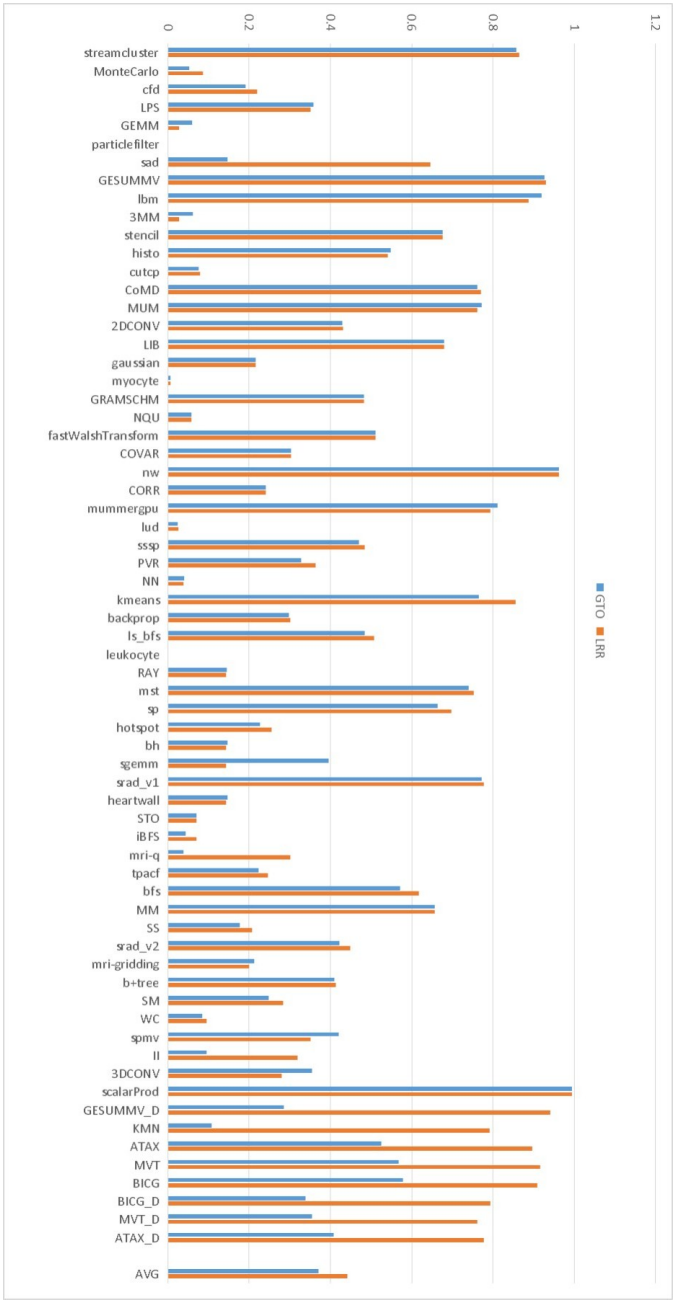


Figure 3.5: Comparing L2 data cache miss rate of Greedy-Then-Oldest (GTO) algorithm and Loose-Round-Robin (LRR) scheduling algorithm

between the GTO algorithm and the LRR algorithm. Similarly, Figure 3.5 compares the L2 miss rate for these two scheduling algorithms. As shown in these figures, the benchmarks that prefer GTO algorithms do so mostly because they get lower L1 and L2 data cache miss rates when the GTO algorithm is applied.These results match prior studies [77] in terms of scheduling algorithm preference. Therefore, we choose the GTO algorithm as the default thread/warp scheduling algorithm in our study. In this dissertation, we advocate that the cache replacement and bypassing algorithm must consider the effect of the thread scheduling algorithm. Based on the scheduling algorithm comparison, we select GTO as the most effective scheduling algorithm. In Chapter 5 and Chapter 6, we develop thrash-resistant and pollution-resistant GPU cache algorithms adapting to the GTO thread scheduling algorithm.

3.5 Methodology

We model the proposed architecture using GPGPU-Sim (version 3.2.1) [1, 4], which is a cycle-level performance simulator of a general purpose GPU architecture supporting CUDA [65]4.2 and its PTX ISA [66]. The GPU simulator is configured to be similar to NVIDIA GTX480 [63] using the configuration file provided with GPGPU-Sim [2]. Compared with NVIDIA GPUs, GPGPU-Sim obtains IPC correlation of 97.3% on Rodinia benchmarks [10]. Key microarchitectural parameters of the baseline configuration are summarized in Table 3.4.

We augment the baseline GPGPU-Sim model with cache set index hashing to improve memory system robustness. We implement set hashing at the

Table 3.4: Baseline GPGPU-Sim configuration.

Number of SMs	15
Threads per SM	1536
Threads per warp	32
SIMD lane width	32
Registers per SM	32768
Shared memory per SM	48KB
Schedulers per SM	2
Warps per schedulers	24
Warp scheduling policy	Greedy-Then-oldest [77]
L1 cache (size/assoc/block size)	16KB/4-way/128B
L2 cache (size/assoc/block size)	768KB/16-way/128B
Number of memory channels	6
Memory bandwidth	179.2 GB/s
Memory controller	Out-of-order (FR-FCFS)

L1 and L2 to better distribute memory accesses among cache banks to mitigate the effect of bank conflicts and reduce "bank camping" where regular access patterns produce excessive contention for a small subset of cache banks. Any additional deviations to the baseline model are described in the context of the architecture sensitivity studies in the related chapters.

Chapter 4

Thread Scheduler Directed Priority-based Cache Allocation

In this chapter, we enhance the GPU thread scheduler with a mechanism called Priority-based Cache Allocation (PCAL) to address inter-thread L1 cache thrashing and the chip resource saturation problem.

First we motivate this research by investigating how the parallelism that a GPU supports affects the throughput and the chip resource utilization. We demonstrate that throttling techniques [39] [77] are able to reduce the cache miss rate and improve the overall performance, however throttling techniques leave memory bandwidth and other chip resources significantly under-utilized. We observe that the underutilized resources could be sufficient to support extra threads without further polluting the cache.

Next, based on the observations, we propose PCAL to alleviate the L1 cache thrashing problem and the memory system resource saturation. The PCAL-enhanced scheduler separates the concerns of the cache thrashing and the chip resource saturation problem by allowing the total number of threads to be be different with the number of threads to allocate the cache. PCAL exploits the available performance improvement with two optimization strate-

gies, either increasing Thread-Level Parallelism (TLP) while maintaining cache hit ratio, or optimizing cache hit ratio while maintaining TLP. PCAL assigns tokens to threads to indicate their privilege to allocate space in the cache, and gives preferential access to the cache and other on-chip resources to a subset of the threads (i.e. Token-holder threads), allows another subset of threads (i.e. Non-token holder threads) to bypass the caches, and throttles the remaining threads to avoid additional memory access latency. This approach reduces the cache thrashing problem and effectively employs the chip resources that would otherwise go unused by a pure thread throttling approach [39] [77].

Next, we describe the mechanisms for implementing PCAL. We introduce an initial implementation of this technique with a static optimization strategy and parameter selection, called static PCAL. We develop a priority-based cache management strategy tailored to the needs of massively threaded processors with limited per-thread cache capacity. Priority tokens are assigned to warps and grant priority to perform various cache actions, including allocation (fill) and replacement (eviction). We describe how priority tokens can be used to influence caching and scheduling policy. We describe mechanisms and policies for assignment, transfer, and release of these cache priority tokens.

Next, we introduce a variant of PCAL with a dynamic parameter selection mechanism, called dynamic PCAL, to adaptively determine the parameters of PCAL . Dynamic PCAL monitors the key statistics of memory system including cache miss ratio, memory pipeline stall, Miss Status Holding Registers (MSHR) reservation failure, L1 block reservation failure, Network-

on-Chip (NoC) transmission latency. Based on this statistics, it dynamically determines the number of threads to allocate the cache and the number of threads bypassing the cache.

We then describe several extensions to the baseline design, such as optimization bypassing traffic by only fetching a portion of the cache block and applying PCAL to the L2 in addition to the L1. At the end, we analyze the experimental results, and observe 67% improvement over the original as-is code and a 18% improvement over a better-tuned warp-throttling baseline.

4.1 Understanding How Parallelism Affects Caches Hit Ratio, Chip Resource Saturation and Throughput

GPGPU programming models, such as CUDA [65] and OpenCL [3], encourage the programmers to expose a large amount of parallelism. However GPU often can not achieve a high occupancy due to control divergence [16] [17] [57] [90] [59] [72] [74] [73] [95] and insufficient memory throughput [57] [39] [77]. This work aims to improve cache efficiency to achieve higher GPU memory system throughput. Mapping the maximum amount of parallel work that the hardware can support often leads to cache thrashing and chip resources saturation problem and thus does not necessarily ensure the best overall performance.

Figure 4.1 shows the chip resources that a GPU memory system integrates, including the L1 and L2 cache blocks (❶ and ❹ in Figure 4.1), the L1 and L2 MSHR tables (❷ and ❺ in Figure 4.1), the Network-on-Chip bandwidth

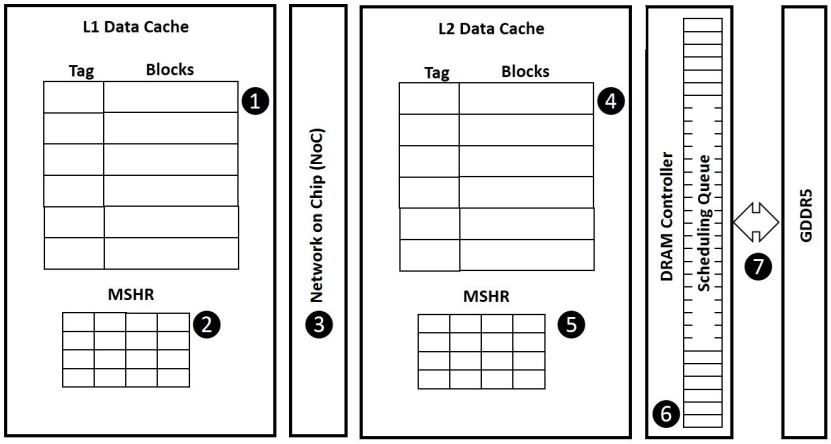


Figure 4.1: Chip Resources in a GPU Memory System.

(❸ in Figure 4.1), the DRAM controller scheduling queue (❻ in Figure 4.1) and the Off-chip bandwidth (❼ in Figure 4.1). A memory request, depending on whether it hits the L1 or L2 cache, consumes a subset of these chip resources to fetch a data block from the GPU memory system. For instance, when a load request misses both the L1 and L2 caches, in order to fetch a data block from the main memory, it needs to reserve a L1 cache block, a L1 MSHR table entry, a L2 cache block, a L2 MSHR entry and a DRAM controller scheduling queue entry. It consumes both the NoC bandwidth and the off-chip bandwidth. If the resources are already in use, the request stalls until the resource become available again.

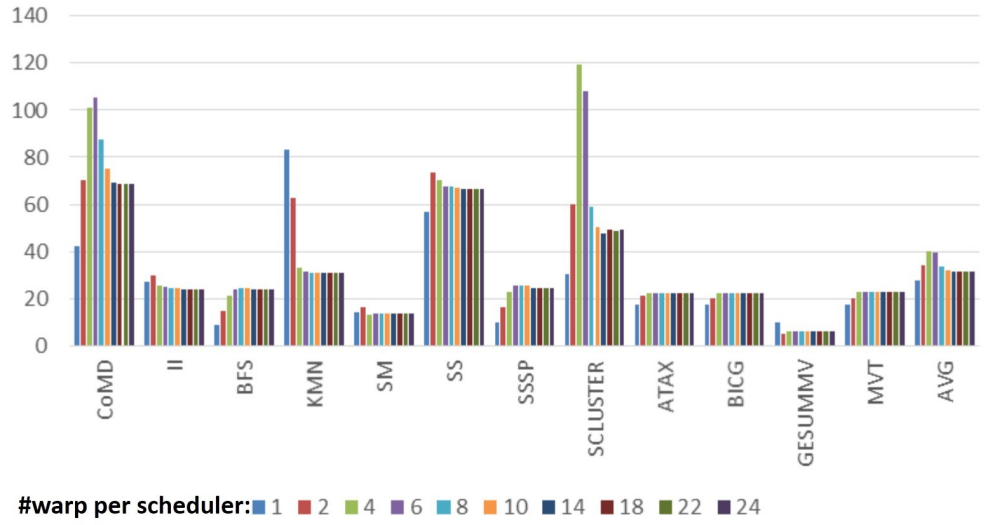
To investigate how parallelism affects the cache hit ratio, memory system utilization and the GPU performance, we characterize a set of cache-sensitive applications by statically varying the maximum number of warps that each thread scheduler allows. Our baseline GPU, simulating a contem-

porary Fermi GPU, allows up to 24 warps per scheduler. When we limit the number of warps per scheduler, we limit the number of warps that a scheduler can select instruction from.

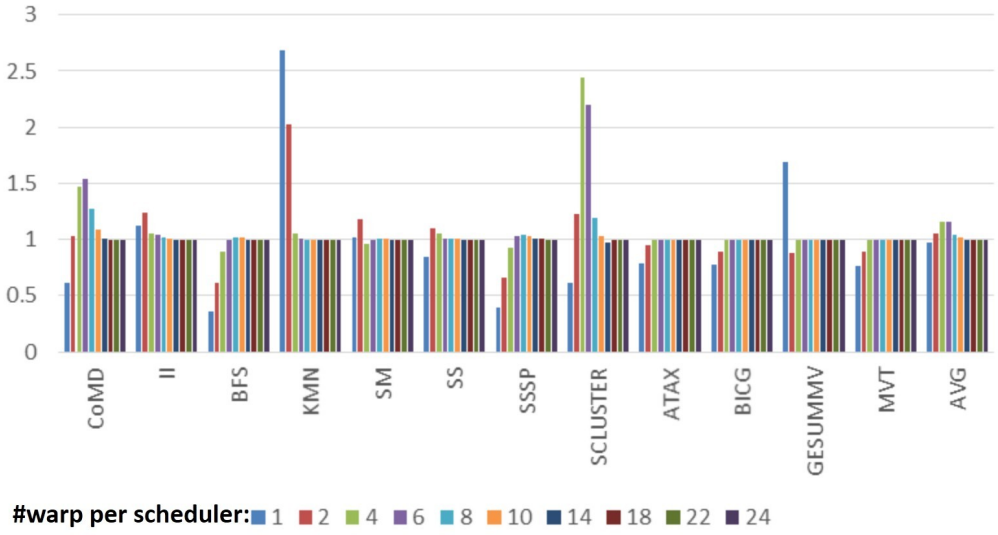
4.1.1 Throughput

Figure 4.2 shows how the Instruction-Per-Cycle (IPC) and the speedup vary with hardware-resident warp count per scheduler for a subset of the benchmarks. For many applications, running with the maximum thread-level parallelism that the hardware can support does not result in the best application performance. For instance, CoMD performs best with only 6 of 24 warps enabled. Others, like KMN, show no benefit from additional warps beyond one.

To establish a firmer baselines for comparison, we define `warp-max` as the maximum number of warps that fit per scheduler and `warp-opt` as the number of warps per scheduler that provides the best performance. The value of `warp-max` is not only limited by the number of threads a SM supports, but also by the other resource constrains such as the total share memory size that all threads declare can not exceed the share memory capacity that the GPU supports. Roughly, `warp-max` represents naive code and `warp-opt` represents software better tuned to our machine configuration. Table 4.1 summarizes the default (max) warp count and performance-optimal warp count.Compared to the `warp-max` configurations, the speedups that `warp-opt` configurations provide are also summarized in the table.



(a) IPC



(b) Speedup

Figure 4.2: IPC and Speedup. Varying maximum number of warps per scheduler.

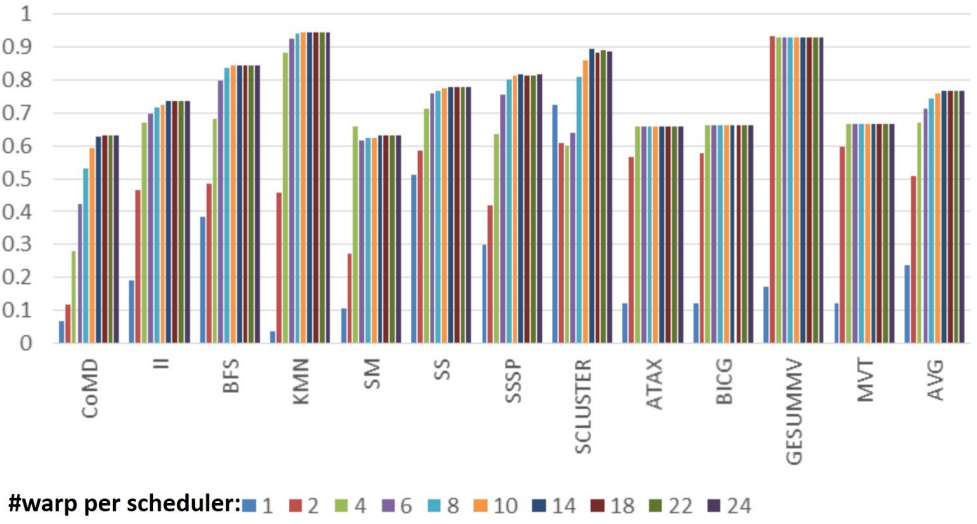
Table 4.1: `warp-max` and `warp-opt` for key benchmarks

application	`warp-max`	`warp-opt`	`Speedup`
CoMD	16	6	1.53
II	24	2	1.24
BFS	24	8	1.02
KMN	24	1	2.68
SM	24	2	1.18
SS	24	2	1.11
SSSP	24	8	1.05
SCLUSTER	24	4	2.44
ATAX	4	4	1
BICG	4	4	1
GESUMMV	4	1	1.69
MVT	4	4	1

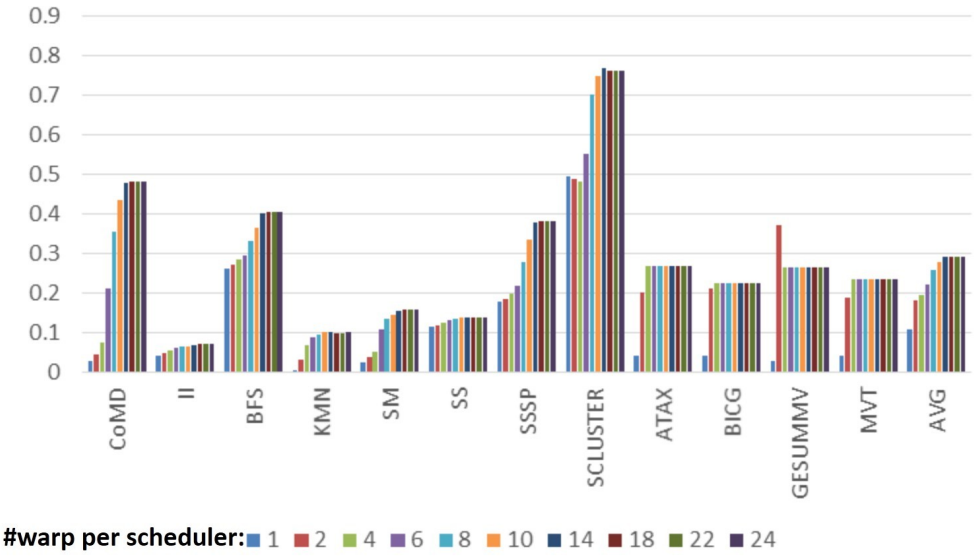
.

4.1.2 Cache Miss Ratio

Figure 4.3a and Figure 4.3b demonstrate how the local L1 miss rate and the global miss rate (ratio of memory requests that miss both L1 and L2 cache) vary with hardware-resident warp count per scheduler. When warp count per scheduler increases from one to maximum, the total TLP increases. However, for many applications, both the L1 and global miss rate increase dramatically with a higher TLP. In this case, the cache thrashing prevents the L1 data cache from capturing most data reuse. The overall cache miss ratio also increase significantly with a higher TLP. For instance, the L1 miss rate of CoMD increases from 7% to 64% when warp count per scheduler increases from one to eight. The global miss rate of CoMD increase from 3% to 48%,



(a) L1 miss ratio



(b) Global miss ratio (L1 miss ratio * L2 miss ratio)

Figure 4.3: L1 and global cache miss ratio. Varying maximum number of warps per scheduler.

showing a very similar trend as L1 miss rate with a higher TLP. The optimal warp count per scheduler is a trade-off between overall TLP and cache miss ratio. For example, CoMD performs best with 6 warps per scheduler, which leads to neither highest TLP nor lowest cache miss ratio.

4.1.3 Memory Request Latency

When a resource in the GPU memory system is saturated by the outstanding memory requests, the memory requests can not move forward until the resource becomes available again. This saturation-induced latency increases dramatically with a higher TLP. We measure this effect with the average round-trip latency of the memory requests that miss both the L1 and L2 cache. The round-trip latency of a memory request is defined as the number of cycles after the request misses the L1 cache for it to go through the memory hierarchy to fetch the data block from the main memory and come back to the SM.

Figure 4.4 shows the average round-trip latency of all memory requests when varying the number of warps per scheduler. The minimal round-trip latency is about 200 cycles. However, when the total TLP increases, some benchmarks experience dramatic round-trip latency increases. For instance, when the TLP increases, the round-trip latency of CoMD increases from less than 200 cycles to more than 600 cycles. In this case, the memory system gets saturated and becomes a performance bottleneck of the GPU.

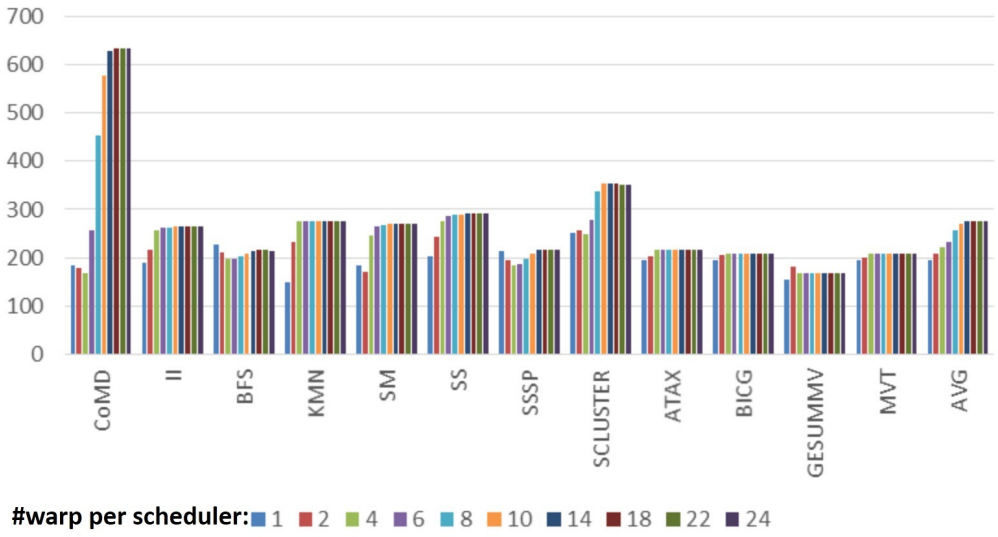


Figure 4.4: Round-trip latency of memory requests. Varying number of warps per scheduler.

4.1.4 Chip Resource Utilization

When the TLP increases, the GPU memory system throughput degrades for two reasons: (1) cache thrashing, which we discussed in Section 4.1.2, (2) chip resource saturation, which often leads to a large memory latency increases. These two problems often happen simultaneously. Cache thrashing leads to more outstanding requests and more resource saturations, which in turn increases memory latencies and activates more memory requests to further thrash the cache. In this section, we measure the NoC latency, the memory pipeline stall ratio, the memory partition congestion ratio and the off-chip bandwidth utilization, to identify the most severe bottlenecks.

NoC latency

We define the NoC latency as the sum of the average sending latency and the average receiving latency of all packets transmitted through the NoC. When the NoC bandwidth becomes a bottleneck, the NoC latency exhibits significant increases.

Figure 4.5 shows the NoC latency of all memory requests when varying the number of warps per scheduler. When the NoC is not saturated, the NoC latency in general is smaller than 20 cycles. For instance, when no throttling technique is applied, the NoC latency of most applications is not greater than 20 except CoMD, KMN, SS and SM. Even for these four benchmarks, the NoC latency is less than 20 when the maximum warp count per scheduler is one. However, when the total TLP increases, CoMD, KMN, SS and SM experience dramatic NoC latency increases. There are three reasons for the NoC latency increase when TLP is higher. Not only are there more active threads, but also the per-thread cache miss ratio increases dramatically. Additionally, other saturated resource could propagate a back pressure signal to the NoC output to keep the requests in the NoC from transmitting to the next stage, and thus causes the NoC latency increase.

Memory pipeline stall ratio

For a thread scheduler, a memory pipeline stall cycle happens when the scheduler finds at least one ready memory instruction to issue, but the memory pipeline is stalled. The memory pipeline stall ratio is the average ratio across

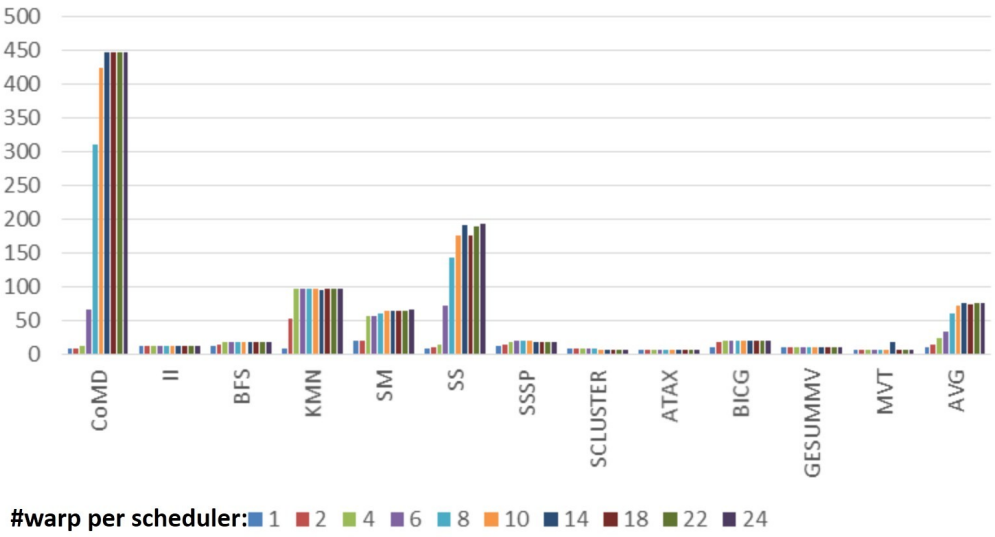


Figure 4.5: Network on Chip (NoC) latency. Varying the maximum number of warps per scheduler.

all schedulers in all SMs of overall cycles when the scheduler finds ready memory instructions but fails to issue them because of a memory pipeline stall. Memory pipeline stalls happen mainly because of back pressure from the L1 cache. When a memory request misses the L1 cache, it needs to first reserve a L1 cache block. Next, it reserves a MSHR entry if there are no previous outstanding memory requests pending on the same memory block. Failing to reserve the cache block or the MSHR causes the load-store unit to hold the instruction and stall the memory pipeline.

Figure 4.6 shows the memory pipeline stall ratio of the key benchmarks when varying the number of warps per scheduler. For this warp-max GPU configuration, the benchmarks experience 51% memory stall cycles on aver-

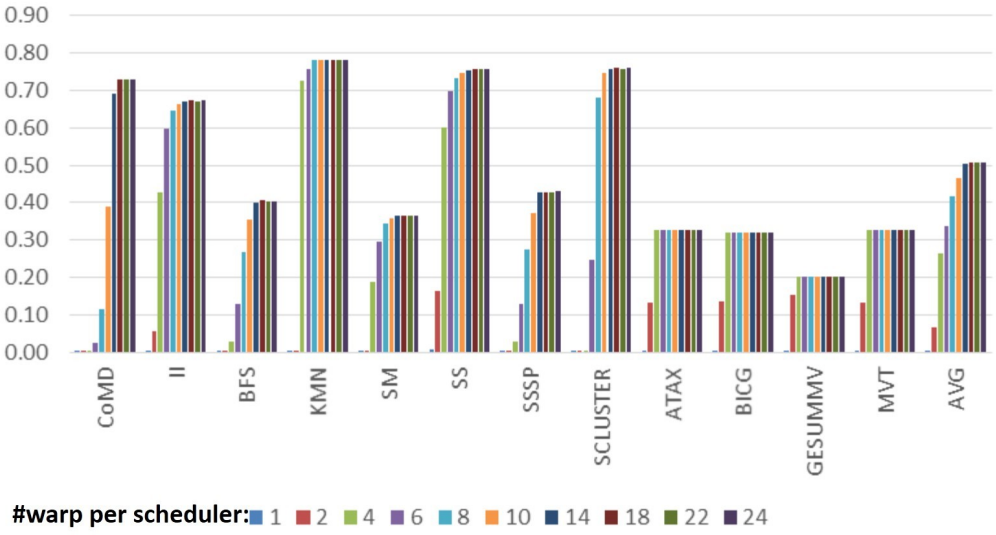
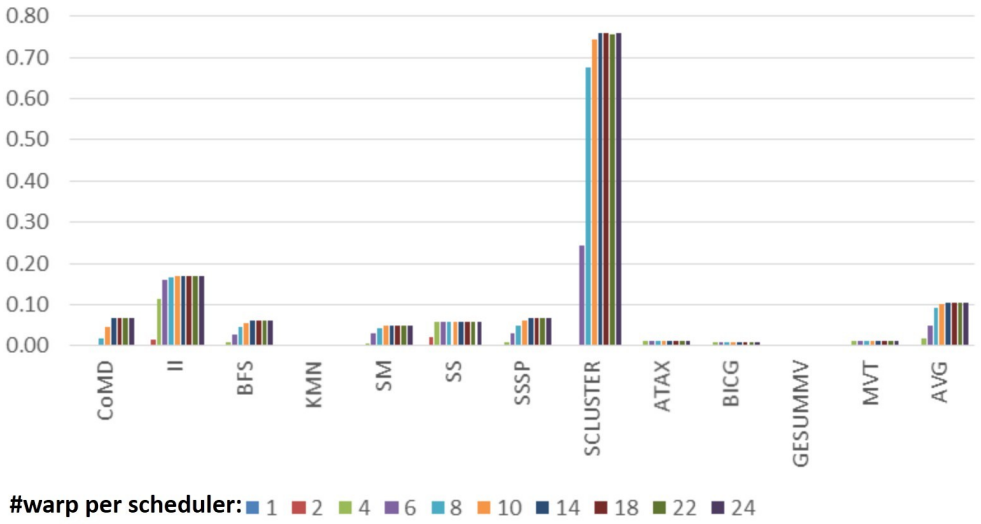


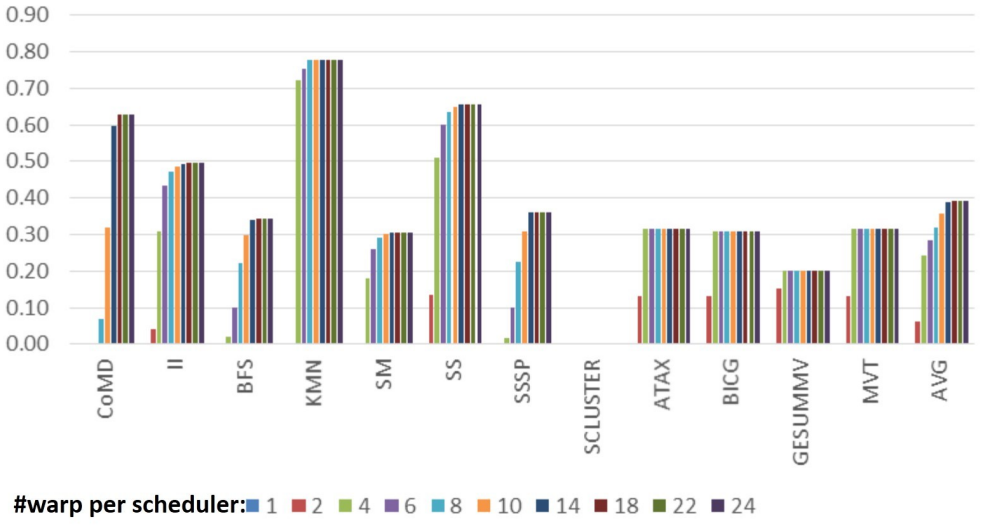
Figure 4.6: Memory pipeline stall ratio. Varying the maximum number of warps per scheduler.

age. More than half of all the execution time, some resources in the memory system are saturated and propagate the back pressure to the warp scheduler so that the scheduler can not issue any instruction. For these memory-intensive benchmarks, significant opportunities exist to manage the parallelism explicitly to avoid unnecessary delays caused by any chip resource saturation and thus improve the memory system throughput significantly.

Figure 4.7 breaks down the memory pipeline stall ratio by the cause of the stall. Figure 4.7a shows memory pipeline stalls ratio due to the reservation failure of the L1 cache block. Figure 4.7b demonstrates memory pipeline stalls ratio due to the reservation failure of the MSHR entry. For most of the benchmarks except SCLUSTER, MSHR reservation failure is the major cause



(a) Memory pipeline stall caused by the L1-Block Reservation Failure



(b) Memory pipeline stall caused by MSHR Reservation Failure

Figure 4.7: Breaking down the ratio of memory pipeline stall by its two causes: L1-Block Reservation Failure, MSHR Reservation Failure. Varying maximum number of warps per scheduler.

of memory pipeline stalls. When the baseline configuration is applied, the schedule allows up to 24 warps to be scheduled, The L1-Block caused memory pipeline stall ratio is mostly lower than 15%. SCLUSTER is an exception, most of its memory pipeline stalls are caused by the L1 block reservation failure. The reason is that SCLUSTER experiences more conflict misses. Before it reserves all MSHR entries, some sets of the cache are accessed more often and lead to reservation failure of the L1 blocks.

For most applications, MSHR reservation failures happen more often than L1 block reservation failures. The major reason is that the baseline GPU configuration enables 128 L1 cache blocks and 32 MSHR entries. Each MSHR entry allows up to 8 requests pending on the same entry. For most of the benchmarks, each MSHR entry only has one pending request for most of the time. In this case, the MSHR table can hold up to 32 outstanding requests, while the number of L1 blocks is 128, it is four times bigger than the number of outstanding memory requests that the MSHR table can practically hold. Consequently, the L1 MSHR reservation failure happens more often than the L1 block reservation failure.

Memory partition congestion ratio

When a memory request transmits to the NoC output, it applies to enter a memory partition which consists of the L2 banks and the DRAM controller. The memory partition may respond with a *busy* signal to indicate it can not accept new request at the time. We define the memory partition congestion

ratio to be the average ratio across all memory partitions of overall cycles when the interconnect outputs to DRAM channel is congested. A high memory partition congestion ratio means some resources in the memory partition are saturated. These resources include the L2 cache blocks, MSHR, DRAM controller scheduling queue or the off-chip memory bandwidth.

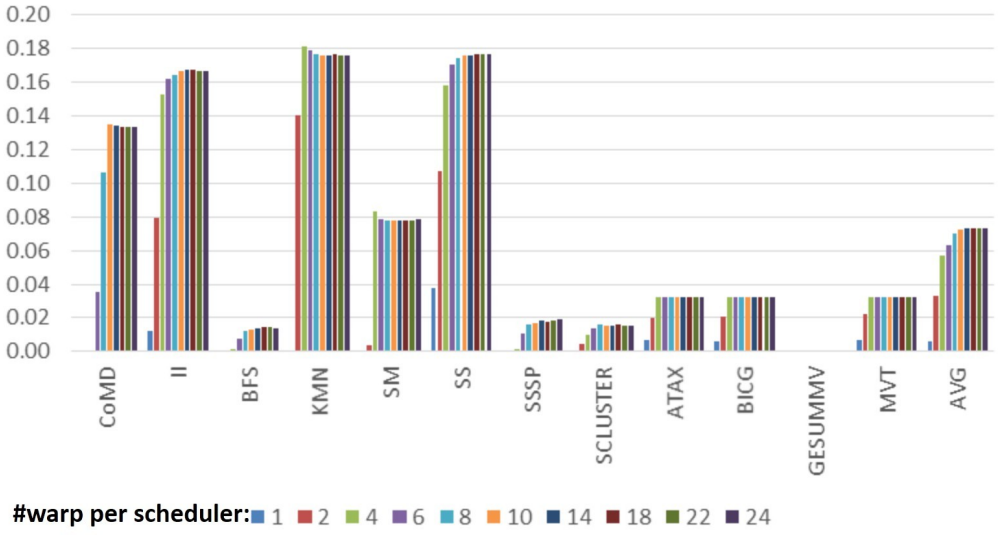


Figure 4.8: Memory partition congestion ratio. Varying the maximum number of warps per scheduler.

Figure 4.8 shows that the memory partition congestion ratios of the key benchmarks when varying the number of warps per scheduler. Most benchmarks, except CoMD, II, KMN and SS, show less than 5% memory partition congestion. This indicates that these benchmarks are not bottlenecked by the resources related to memory partition, such as off-chip bandwidth or L2 MSHR table etc. For the other four benchmarks, CoMD, II, KMN and SS, the

memory partition congestion ratio is also lower than 5% when the schedule only allows one warp to be scheduled. However, when TLP is higher, this congestion ratio increases significantly. They all show more than 10% memory partition congestion when the baseline configuration (24 warps per scheduler) is applied. The congestion ratio increase is caused by both L1/L2 thrashing with the high TLP.

Off-Chip bandwidth utilization

DRAM controllers communicate with DRAM chips through the off-chip bus. The ratio of overall cycles when the off-chip bus is busy is referred to off-chip bandwidth utilization. A low off-chip bandwidth utilization does not necessarily means the off-chip bandwidth has not been a bottleneck during the execution. When the memory requests become bursty, the off-chip bandwidth becomes saturated in a short period. However, the busy period might only be a small portion of overall execution time. The overall off-chip utilization might still be low.

Figure 4.9 shows how the off-chip bandwidth utilization varies with hardware-resident warp count per scheduler. For most applications, the bandwidth utilization increases with the higher TLP. The reasons are two-fold. First, higher TLP leads to more outstanding memory requests and requires more memory bandwidth. Second, higher TLP results in the higher cache miss ratios and thus more data needs to be fetched from off-chip DRAM modules. When a throttling technique selects the best warp count per scheduler,

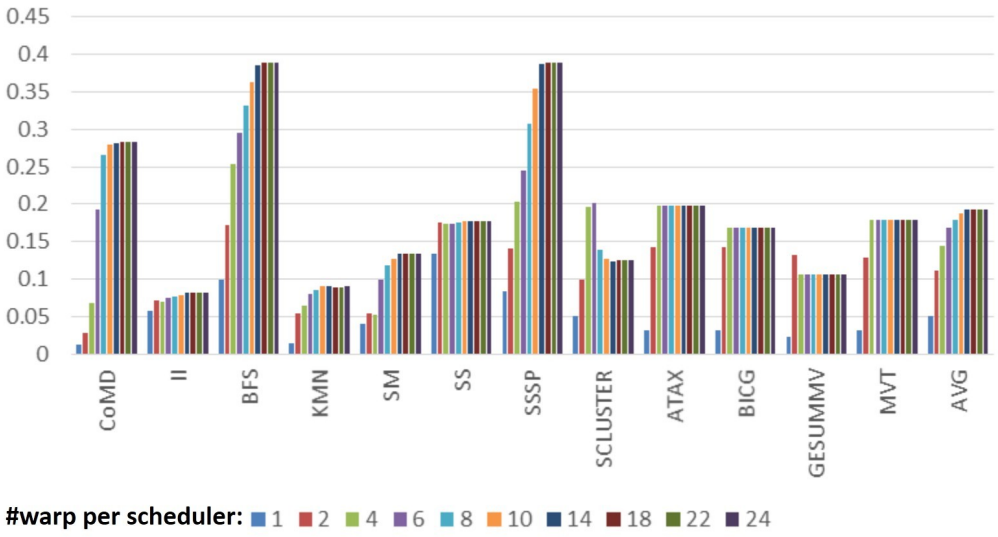


Figure 4.9: Off-Chip Bandwidth Utilization of benchmarks. Varying the maximum number of warps per scheduler.

the maximum TLP is often not enabled.

Summary

As we have discussed in this section, memory-intensive applications stress the GPU memory system extensively. Cache thrashing and chip resource saturation happen simultaneously. On one hand, when the TLP is higher, the cache thrashing problem leads to more outstanding memory requests. More memory requests require more storage to keep the data blocks and the meta data and more on-chip/off-chip bandwidth to transmit the requests and the data blocks. When the related chip resources get saturated, the memory request round-trip latency increases significantly. The thread scheduler activates more threads to find more instructions to hide the memory request latency.

We observe that the number of threads that thrash the cache is often different than the number of threads that saturate the chip resources. Cache thrashing is only affected by the cache access patterns. It happens when most reuse distances exceed the cache capacity. Chip resource saturation is not only caused by the access pattern, but also affected by the ratio of the memory instructions(i.e. memory intensity). For instance, if an application travels a 1-million node link-list in order for 10 times, our baseline GPU can not capture the locality because the average reuse distance of memory requests is much larger than the cache capacity. Consequently, the cache is thrashed. However, if the memory intensity of this application is $\frac{1}{1\,million}$, there would not be many active outstanding memory requests. Consequently, chip resource is not likely to be saturated.

It is desirable to separate the concerns for the cache thrashing and the memory system resource saturation. When the cache is thrashed but the chip resources are underutilized, new mechanisms are needed to utilize the resources more effectively without further thrashing the cache.

4.1.5 Identifying Performance Bottleneck

In this section, we analyze the performance, the L1/L2 cache miss ratio and the chip resource utilization metrics varying maximum TLP allowed on each scheduler. We identify the major performance bottlenecks for each application when TLP is higher. Applications are grouped into categories based on the bottlenecks.

The cache thrashing and the chip resource saturation problem happen simultaneously, most of these memory-intensive applications not only thrash the cache but also saturate some chip resource. The saturation of different chip resources also affect one other. For instance, when the NoC bandwidth becomes a bottleneck, the memory access latency increases and thus the memory requests reserve the cache blocks and the MSHR entries longer. As a result, the NoC saturation problem increases the number the cache block and MSHR reservation failures and thus causes more memory pipeline stalls. Therefore, an application can have more than one bottleneck at the same time.

We analyze application bottlenecks as follows.

- **Step 1: Parallelism starvation**

  For a given application, we compare `warp-opt` and `warp-max`. If they are equal, it means the application reaches the highest performance with the highest TLP. Although we can not determine whether the performance can keep increasing with a TLP higher than `warp-max`, we identify the bottleneck of the application to be parallelism starvation.

- **Step 2: Chip resource saturation**

  The chip resource saturation problem puts a hard limit for the GPU throughput. To investigate why performance degrades when TLP exceeds `warp-opt`, we compare the statistics when the TLP equals `warp-opt` $+\Delta$ and the ones when the TLP equals `warp-opt`, where `warp-opt` $+\Delta$ is the minimum warp number profiled that is bigger than `warp-opt`. For

instance, the `warp-opt` value of KMN is one and the `warp-opt` $+\Delta$ value of KMN is two.

  - **Step 2.1: Memory partition congestion ratio**

    If the memory partition congestion ratio increases more than 30% and exceeds 20%, we identify memory partition congestion as one of the bottlenecks of the benchmark.

  - **Step 2.2: NoC latency.**

    If the NoC latency increases more than 30% and exceeds 20%, we identify the NoC bandwidth as one of the bottlenecks of the benchmark.

  - **Step 2.3: MSHR reservation failure.**

    If the memory pipeline stall ratio due to L1 MSHR reservation failures increases more than 30% and exceeds 20%, we identify the L1 MSHR reservation failure as one of the bottlenecks of the benchmark.

  - **Step 2.4: L1 block reservation failure.**

    If the memory pipeline stalls due to the L1 block reservation failures increases more than 30% and exceeds 20%, we identify L1 cache blocks reservation failure as one of the bottlenecks of the benchmark.

- **Step 3: Cache thrashing**

  For a given application, we compare the L1 miss ratio when the TLP

equals `warp-opt` and the one when the TLP is equal to one. If the normalized number of the cache misses increases more than 2X and the cache miss ratio exceeds 30%, we identify cache thrashing as the bottleneck of the benchmarks.

We apply these steps to our benchmark set. Table 4.2 summarizes the major bottlenecks of the benchmarks. The performance bottlenecks, such as the cache thrashing, the L1 block/MSHR reservation and the NOC bandwidth saturation problems happen simultaneously. It is common that some benchmarks exhibit multiple performance bottlenecks. These bottlenecks are specific to the application running on this GPU configuration. Changing the GPU configuration can change the bottlenecks of the benchmarks.

4.2 Motivation: Two Performance Opportunities Beyond Throttling

In this section, we motivate our research based on our bottleneck analysis. We first explain why throttling the total number of threads can improve the overall performance. Next, we discuss two opportunities that can further improve the performance beyond the known throttling techniques.

4.2.1 Understanding Throttling Techniques

The massive amount of parallelism that GPU hardware supports has both positive and negative effects on GPU performance. On one hand, a higher TLP enables a GPU to tolerate more long latency operations by fast context

Table 4.2: Identifying the bottlenecks for key benchmarks when increasing TLP beyond `warp-opt`. The acronyms are defined as follows, `ParStrv`: Parallelism Starvation, `MemCong`: Memory Partition Congestion, `NoC`: NoC Bandwidth Saturation, `L1MSHR`: L1 cache MSHR reservation failure, `L1Blk`: L1 cache block reservation failure. `L1Thrash`: L1 cache thrashing problem

Application	`ParStrv`	`MemCong`	`NoC`	`L1MSHR`	`L1Blk`	`L1Thrash`
CoMD			Y			Y
II				Y		Y
BFS				Y		Y
KMN			Y			Y
SM			Y			Y
SS				Y		
SSSP				Y		Y
SCLUSTER					Y	
ATAX	Y					Y
BICG	Y					Y
GESUMMV						Y
MVT	Y					Y

switch among threads. On the other hand, the TLP affects L1 miss rate and L2 miss rate dramatically. When the maximum warp count per scheduler is allowed, both cache thrashing and resource saturation often become major problems that degrade overall throughput.

Application performance can be improved by throttling the TLP from `warp-max` to `warp-opt`. The optimal warp count per scheduler is a trade-off between overall TLP, cache miss ratio and degree of resource saturation. For example, CoMD performs best with 6 warps per scheduler, which leads to neither the highest TLP, nor the lowest cache miss ratio nor the lowest resource congestion ratio.

4.2.2 Two Performance Opportunities Beyond Throttling

Although the throttling techniques can improve the overall throughput for some applications, they rely on tuning only one parameter, the total number of threads, to balance the total TLP, the cache miss ratio and different resources in the memory system. The trade-often is often sub-optimal. As we observe, there are two opportunities to improve the performance beyond the throttling techniques.

1. **Increasing parallelism without thrashing caches**

   When the TLP is higher, cache thrashing and the reservation failure of L1 block/MSHR have negative effects on performance. The throttling techniques balance the overall performance among the TLP, the cache miss ratio and cache resource reservation. When TLP is equals to

`warp-opt`, many chip resources are still underutilized, such as the NoC bandwidth and the off-chip bandwidth etc.

For example, SCLUSTER performs best with 4 warps per scheduler. As shown in Table 4.2, its major bottleneck is the L1 block reservation failure. Both the NoC bandwidth and the off-chip bandwidth are not limiting the performance. If extra threads beside the 4 warps can be enabled without reserving more L1 blocks, the performance of the original 4 warps should not decrease. The extra threads can utilize the spare NoC bandwidth to reach L2, and utilize the spare off-chip bandwidth to access the main memory. Consequently, the extra threads can achieve extra throughput beyond the throttling techniques.

2. **Increase cache hit ratio without decreasing TLP**

   Throttling techniques allow all threads to access the cache. When optimal throttling is applied, many benchmarks already experience the cache thrashing problem. There is opportunity to alleviate this thrashing problem. If a subset of threads have higher priority to utilize the cache, their working sets might fit in the cache to get a high cache hit ratio. Although the other threads experience a lower cache hit ratio, the overall cache hit ratio might be higher.

4.3 Token-based Prioritized Cache Allocation (PCAL)

To address the cache thrashing and the chip resource saturation problem of many-threaded GPGPU architectures, throttling techniques have been proposed to select the optimal number of threads to alleviate these problems.

However, trashing techniques rely on tuning only one parameter, the total number of threads, to balance the total TLP, the cache miss ratio and different resources in the memory system. As we discuss in Section 4.2, the trade-¿offs that throttling techniques make are often sub-optimal. We develop *Priority-based cache allocation* (PCAL), to separate the concerns of the cache thrashing and resource saturation problems, to exploit the two opportunities we discuss in Section 4.2.2. PCAL is orthogonal to previous thread-throttling mechanisms as it can synergistically be adopted on top of previous schemes. The warp-level-throttling can tune the overall all parallelism at finest granularity to achieve better performance than CTA-level-throttling. Therefore, we conduct the experiments of PCAL on top of optimal warp-level throttling.

The key idea of PCAL is that the thread scheduler identifies the performance bottleneck, then categorizes all memory references to two groups by warps: (1) the cache-thread group that allocates cache capacity. (2) the bypassing-thread group that needs to bypass cache. The cache controller on the other hand, gives the cache threads preferential access to the cache. The bypassing threads are forced to bypass the caches. We use *tokens* to represent priorities in the memory hierarchy (cache system). Only the memory references from the cache thread group are token-holders, the remaining ones are

the non-token holders.

4.3.1 Strategies and Challenges

Two Strategies

As we discuss in Section 4.2.2, there are two basic strategies to improve the overall performance. An application might be feasible for one, both or none of them. PCAL exploits both of the two opportunities as follows.

1. **Strategy one: Increasing TLP while maintaining cache hit ratio**
   PCAL keeps the number of the cache threads equal to `warp-opt`, and selects the minimum number of extra bypassing threads to saturate the chip resources. In this case, the cache hit ratio of the cache threads does not change, these threads are expected to maintain the same throughput as the throughput with the throttling mechanism. The extra bypassing threads enable PCAL to utilize chip resources more effectively and get extra throughput beyond that provided by the throttling mechanism.

2. **Strategy two: Increasing cache hit ratio while maintaining TLP**
   PCAL keeps total TLP equal to `warp-opt` and reduces the number threads to allocate the cache. The subset of threads that share the cache are the token-holder threads, which keep their working sets in cache. The remaining threads are the bypassing threads which do not thrash the cache.

Key challenges

The major challenges of PCAL are two-fold as follows.

1. **Identify the bottleneck and deciding the basic strategy**

   There are two basic strategies to improve the overall performance. For an application, PCAL needs to choose and configure one or neither of the two strategies adaptively. PCAL collects the statistics such as IPC, cache miss ratio and chip resource usage metrics etc. PCAL investigates the feasibility of strategy one, `increasing parallelism while maintaining cache hit ratio`. PCAL estimates whether the spare resources are sufficient for adding extra bypassing threads. If resource saturations happens, PCAL considers strategy two, `increasing cache hit ratio while maintaining TLP`. If PCAL speculates that neither of the two strategies can help improve the performance, it keeps the application running with the pure throttling technique.

2. **Decide the size of the cache thread group size and the bypassing thread group**

   After PCAL selects the right strategy to improve the performance, it configures the number of the cache threads and the number of bypassing threads. Finding the right parameters is critical for PCAL to achieve higher throughput. For strategy one, PCAL can achieve the best performance when the number of the bypassing threads is the minimum number of extra bypassing threads need to saturate the chip resources.

For strategy two, the overall cache hit ratio can be increased only if the total working set of the cache threads can fit in the L1 cache.

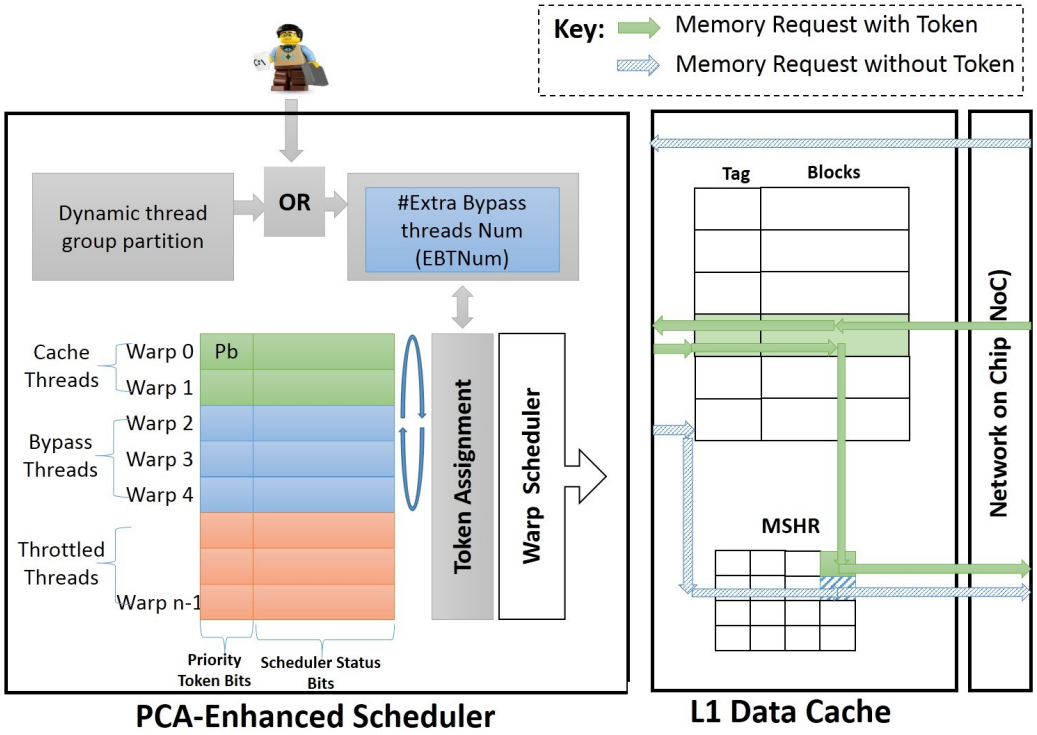


Figure 4.10: Architectural Overview. Paths potentially effected by token priorities are shown.

Initially, we consider software-directed optimization strategy selection and parameter assignment on a per-kernel basis. The size of the thread groups may be selected by the developer or the compiler/auto-tuner/profiler and is supplied to a kernel at launch time with other kernel launch parameters. We also develop a dynamic mechanism, namely `Dynamic PCAL`, which decides the strategy and the size of the thread groups adaptively at runtime.

4.3.2 Static PCAL

In this section, we consider the software-directed static solution, `Static PCAL`. For this research, our baseline is the optimal static warp throttling, which provides `warp-opt` as the number of warps per scheduler that enables the best performance. Static PCAL allows the programmer to choose the strategy from (1)**Strategy one: Increasing TLP while maintaining cache hit ratio** or (2)**Strategy two: Increasing cache hit ratio while maintaining TLP**

To make the programming interface clear, static PCAL requires the programmer to provide only one parameter: the Extra Bypassing Thread Number ($EBTNum$). Static PCAL decodes the parameter as follows: If $EBTNum == 0$, static PCAL limits itself to the pure warp-level-throttling. It does not enable any extra bypassing thread. If $EBTNum > 0$, static PCAL enables strategy one, increasing TLP while maintaining cache hit ratio. static PCAL sets the size of the cache thread group to be `warp-opt` and sets the size of the bypassing thread group to be $EBTNum$. If $EBTNum < 0$, static PCAL enables strategy two, increasing cache hit ratio while maintaining TLP. static PCAL sets the size of the cache thread group to be `warp-opt` + $EBTNum$ and sets the size of the bypassing thread group to be $(-1) * EBTNum$. The calculation is summarized in Table 4.3.

Figure 4.10 illustrates the basic PCAL implementation. PCAL can be enabled or disabled for the L2 cache. The implementation of PCAL in the L2 cache is similar to its implementation in the L1 cache and thus is not shown in the figure. The initial PCAL implementation is applied only at the L1 cache.

Table 4.3: Input and Strategy of static PCAL. Strategy one and two are introduced in Section 4.3.1. EBTNum is the acronym of the Extra Bypassing Thread Number

Input	`Strategy`	`Size of cache thread group (token number)`	`Size of bypassing thread group (non-token thread number)`
$EBTNum == 0$	Warp-Throttling	`warp-opt`	0
$EBTNum > 0$	Strategy one	`warp-opt`	$EBTNum$
$EBTNum < 0$	Strategy two	`warp-opt` + $EBTNum$	$(-1) * EBTNum$

.

In the rest part of the section, if not specified explicitly, we refer PCAL on L1 only as PCAL.

At a high level, PCAL gets the Extra Bypassing Thread Number ($EBTNum$) from either the programmer or the dynamic thread group partition unit. The token assignment unit decodes the $EBTNum$ value into the size of the cache thread group and the size of the bypassing thread group. It is also responsible for allocating tokens to threads/warps. For each warp, beside its normal scheduler status bits, the Priority bits (Pb) are added to indicate the warp's token value. The priority bits are also added to the memory request packet format, as well as per-line storage of priority token meta-data for the L1 cache. Token data may be transmitted with any memory request or message to propagate status. Cache control logic uses token bits to determine which requests may allocate space in the cache. The token assignment unit in the warp scheduler is responsible for allocating tokens. The major components of static PCAL

implementation are as follows.

Strategy Selection and Token count calculation Static PCAL takes $EBTNum$ as an input parameter. The input parameter EBTNum may be selected by the developer or the compiler/autotuner/profiler and is supplied to a kernel at launch time with other kernel launch parameters. Static PCAL calculates the size of cache thread group i.e. Token Number) and the size of the bypassing thread group i.e. Non-Token group size) using the formulas in Table 4.3.

Token assignment: Tokens may be assigned by various mechanisms, including statically by software or dynamically by hardware. Our implementation assigns N tokens to the N oldest warps.

Token Release: Once assigned a token, a warp retains the token until completion. Tokens are released at warp termination and assigned to the oldest warp that doesn't currently hold a token.

Token implementation: A warp scheduler keeps scheduler state bits for each active warp. Tokens assigned to warps are represented as a bit or bits stored along with the scheduler state bits in the SM-level warp scheduler. When a memory request is generated, the appropriate token priority is attached to the request. The memory system (e.g., cache) uses the priority to influence policy.

PCAL provides preferential cache capacity to the token holder threads, and allows other threads to execute with low or no access to the cache. To achieve this goal, PCAL employs the priority tokens to determine caching policies.

Cache allocation policy: When token holders access the cache, they are allowed to allocate cache blocks so that the working data sets of the corresponding threads resident in the cache and can be further reused. The no-token holders are not allowed to fill any cache block to avoid any pollution in our initial implementation.

Cache eviction policy: Similar to the cache allocation policy, In our initial implementation, possessing a priority token indicates a warp has permission to initiate replacement (eviction). While the no-token holders are not allowed to perform eviction.

Hardware implementation overhead In the simplest case, possession of a token (or lack thereof) can be represented by a single bit. Each warp-level scheduler entry needs to track this additional bit. A small amount of logic is required to allocate and manage tokens. Additionally, memory request messages will need to be tagged with this additional data. Usually, several unused "reserved" bits exist in message formats to permit future extensions such as these. Minimal logic overhead is required for managing the assignment of tokens.

4.4 Dynamic Optimization Strategy Selection and Bypassing Threads Count Prediction

In this section, we propose a dynamic thread group partition mechanism with PCAL, *Dynamic PCAL*. As we discuss in Section 4.3.1, PCAL can exploit two opportunities, either increasing TLP while maintaining cache hit ratio or increasing cache hit ratio while maintaining TLP, to improve the GPU throughput beyond the best throttling technique. Static PCAL allows the programmer to set just one parameter, the Extra Bypassing Thread Number ($EBTNum$). Both the strategy and the size of thread groups can be decided as shown in Table 4.3. Dynamic PCAL chooses the strategy and $EBTNum$ for each application adaptively at runtime.

Section 4.4.1 first motivates the needs of a dynamic PCAL scheme. Next, the high level implementation of the scheme is described in Section 4.4.2. Finally, the detailed algorithm is explained in Section 4.4.3.

4.4.1 Motivation for dynamic PCAL

Static PCAL is an effective tool for allowing experienced programmers to tune the performance of GPU applications. The experienced programmers are able to configure the parameter $EBTNum$ to enable higher throughput. However, finding the best value of the EBTNum requires programmers to have a deep understanding of GPU microarchitectures. It is challenging for normal programmers to apply static PCAL, especially for applications where the best EBTNum value depends on input set or specific GPU microarchitectural pa-

rameters. It is desirable for the normal programmers to have a dynamic scheme that can ease their burden. The goal of dynamic PCAL is to determine the value of EBTNum dynamically and achieve speedups similar to those provided by static PCAL.

4.4.2 Overview

At a high level, dynamic PCAL is static PCAL augmented with an Extra Bypassing Thread Number ($EBTNum$) predictor. Figure 4.11 shows the implementation of such a predictor. A EBTNum predictor chooses the right value of $EBTNum$. It requires the predictor not only predicts the best strategy to improve the performance, but also decides the size of the cache thread group and the bypassing thread group. The warp scheduler and the cache manager work as in the static PCAL scheme that the warp scheduler assigns tokens to a set of warps. The cache manager prioritizes the memory accesses with token to reserve and fill cache blocks.

The EBTNum predictor consists of a phase detector (❶ in Figure 4.11), a statistics collector (❷ in Figure 4.11) and a state-machine based control unit (❸ in Figure 4.11). The phase detector triggers the control unit at every new program phase. The control unit chooses the optimization strategy from two options, (1) increase TLP while maintaining cache hit ratio and (2) increase cache hit ratio while maintaining TLP. It also needs to decide the best value of $ENTNum$ for the chosen strategy based on the statistics collected by the statistics collector.

If the control unit finds that there is no sufficient resource to support a bypassing thread, it applies strategy two to optimize the cache hit ratio. For instance, the NoC bandwidth is already saturated or the memory pipeline stalls often due to MSHR reservation failure. In this case, increasing TLP is not a good option to improve the performance. Dynamic PCAL starts to apply the optimization strategy two: removing some threads from cache threads group so that the cache threads can reside their working set in the cache. Dynamic PCAL searches the best number of the cache threads that minimizes the overall cache miss ratio with the parallel voting algorithm. Dynamic PCAL assigns a different value of $EBTNum$ to each SM. Each SM is able to test the performance effect for a particular $EBTNum$ value. Next, the throughput of all tested $EBTNum$ value are collected and compared. The $EBTNum$ value which leads to the highest throughput for the corresponding SM is voted to be the optimal value of $EBTNum$. This value is applied to all schedulers in all SMs.

When the control unit finds that the spare resources are sufficient to support extra bypassing threads, it exploits optimization strategy one to increase TLP while maintaining cache hit ratio. Dynamic PCAL searches for the minimum number of extra bypassing threads that saturate the chip resources using a classic hill-climbing algorithm. The control unit increases the value of $EBTNum$ by one at every sampling period until it finds that the new value of $EBTNum$ leads to a lower throughput compared to the performance of the previous sampling period.

The hill-climbing algorithm takes multiple sampling periods to find the target value, while the parallel-voting algorithm normally only needs one sampling period. However, the parallel-voting can not be applied to decide the extra bypassing thread number when the strategy one is applied to increase TLP. The reason is that some chip resources, such as the NoC bandwidth, are shared among all SMs. The saturation of such resources is induced by the total number of the memory requests, not the number of requests from a single SM. Therefore, the parallel-voting algorithm can not identify the best value of $EBTNum$ when the extra bypassing threads saturate the chip resources shared by all SMs.

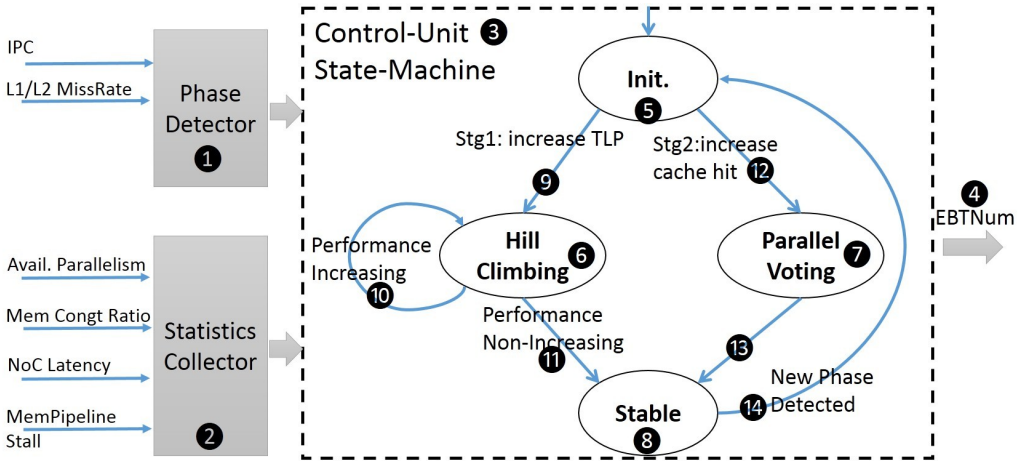


Figure 4.11: Implementation of dynamic PCAL.

4.4.3 Extra Bypassing Thread Number Predictor

The phase detector samples runtime statistics periodically, including L1/L2 cache miss rate and instruction per cycle (IPC). When the relative changing ratio compared to previous sampled values is bigger than a predefined threshold, the phase detector understands either the program has potentially entered a new phase. It identifies a new coming program phase only if the change lasts for several sampling period.

The statistics collector is a collection of programmer counters to diagnose the GPU memory system status. It collects runtime statistics periodically, including the available warps per shader, the memory partition congestion ratio, the NoC latency and the memory pipeline stall ratio, as we have discussed in Section 4.1.4.

The control unit is implemented as a state machine. It is triggered by the phase changing signal, identifies the resource bottleneck based on the statistics, chooses the strategy to improve the performance and decides the value of $EBTNumb$. On a new program phase, the controller first checks the feasibility of the two optimization strategies. If there are sufficient chip resources to increase TLP, the controller applies a hill-climbing algorithm (❻ in Figure 4.11) to find the best number of extra bypassing threads that can be added to the system. It increases $EBTNum$ by one at every sampling period until the performance is no longer increasing. If the controller finds the chip resources are already saturated and are not sufficient to support a higher TLP, it applies the parallel-voting algorithm (❼ in Figure 4.11), exploiting the

possibility to increase the cache hit ratio.

The state, the transition conditions and the actions of the state machine are introduced as follows.

- **Init State**(❺ in Figure 4.11). This state indicates the program is at the beginning of a new program phase. When an application starts, this is the default state. No optimization is applied and the application runs with a default configuration. For instance, when the warp-throttling is applied as the baseline, the application runs with TLP = `warp-opt`. The statistics collector samples the chip resource utilization to allow the control unit to estimate whether spare resources are sufficient to support extra bypassing threads.

  Transition After the application stays at this state for at least one sampling period, the state machine moves to the next state by following rules:

  - **When increasing TLP is feasible (❾ in Figure 4.11), it moves to the Hill-Climbing State (❻ in Figure 4.11)**
    When the following three conditions are all satisfied, the controller predicts that there are sufficient spare resources to support extra bypassing threads. It allows increasing the TLP to improve the throughput. (1) the NoC latency is not higher than a predefined threshold. (2) the memory pipeline stall ratio is not higher than a

predefined threshold. (3) there are warps throttled so that adding TLP is possible.

Action before transition: The controller unit starts to apply the hill-climbing algorithm to search an optimal positive value for $EBTNum$, it sets $EBTNum = 1$ before moving to the Hill-Climbing state.

– **When increasing cache hit ratio is preferred (❿ in Figure 4.11), it moves to the Parallel-Voting State (❼ in Figure 4.11)**

When the controller predicts that there is not sufficient spare resource to support a bypassing thread, it tries to optimize the cache hit ratio to avoid the thrashing.

Action before transition: The controller unit applies the parallel voting algorithm to search for an optimal negative value for $EBTNum$. Initially, before moving to the Hill-Climbing state, the controller assigns a different value of $EBTNum$ to each SM. Therefore, each SM is able to test the performance effect for a particular value of $EBTNum$. The predictor does not have to test all possible values of $EBTNum$, instead it can test every other token number to minimize the test period.

- **Hill-Climbing State**(❻ in Figure 4.11). This state indicates that the controller is applying the hill-climbing algorithm to search for an optimal positive value for $EBTNum$. At this state, $EBTNum$ is the positive

number that decides the number of extra bypassing threads that the spare resources can maintain without saturation happening.

The controller compares the performance of the current sampling period to the performance of the previous sampling period to figure out whether $EBTNum$ needs to increase.

Transition The state machine moves to the next state using following rules.

- $IPC_{current} > 110\% * IPC_{lastSampling}$**, It stays at Hill-Climbing State (❻ in Figure 4.11)**

  It means that enabling more bypassing threads leads to a higher performance. The controller predicts TLP needs to be higher. It increases the bypassing thread number again.

  Action before transition: It increases $EBTNum$ by one.

- $IPC_{current} < 97\% * IPC_{lastSampling}$**, It moves to the Stable state (❽ in Figure 4.11)**

  It means that enabling more bypassing threads leads to a lower performance. The controller predicts TLP needs to be lower. It removes one bypassing thread

  Action before transition: It decreases $EBTNum$ by one.

- $97\% * IPC_{lastSampling} < IPC_{current} < 110\% * IPC_{lastSampling}$**, It moves to the Stable state (❽ in Figure 4.11)**

It means that enabling more bypassing threads does not lead to a significant performance change. The controller considers current TLP to be optimal. The current $EBTNum$ value is kept until a new programing phase comes.

Action before transition: None.

- **Parallel Voting State**(❼ in Figure 4.11). This state indicates that the controller is applying the parallel voting algorithm to search for an optimal negative value for $EBTNum$ to improve the cache performance. Before the state machine moves to this state, the controller assigns different value of $EBTNum$ to each SM. Each SM tests the performance effect for a particular value of $EBTNum$. After the parallel voting unit tests all preferred token numbers, the best value of $EBTNum$ can be voted out based on any predefined goal, such as minimize energy consumption. In our experiment, the goal to select the best $EBTnum$ is to maximize the IPC. At the end of this sampling period, the throughputs for all tested $EBTNum$ values are collected and compared. The value which leads to the highest throughput for the corresponding shader is voted to be the best value of $EBTNum$.

  Transition the state machine moves to the next state by following rules:

  – **It moves to the Stable state (❽ in Figure 4.11)**

  The $EBTNum$ value which leads to the highest throughput among

all SMs is voted to be the best value of $EBTNum$.

Action before transition:All SMs then feed the best $EBTNum$ value to their warp schedulers.

- **Stable mode**. This state indicates the program is in the middle of a stable program phase. The best value of $EBTNum$ has already been set at the beginning of current phase. The phase detector keeps sampling runtime statistics to identify new phase.

  Transition the state machine moves to the next state only when a new phase is detected.

  Action before transition:All optimizations are disabled. It allows the application to run under the default configuration. $EBTNum$ is set to be zero.

4.5 Bypassing Traffic Optimization

In this section, we propose an optimization of PCAL, namely bypassing traffic optimization (BTO), to reduce the size of the bypassing traffic and save NoC bandwidth.

As we discuss in Section 4.1.5, memory intensive GPU applications sometimes saturate chip resources, which often limits the overall GPU throughput. The PCAL mechanism exploits two strategies to optimize system throughput. The first optimization strategy is to enable extra bypassing threads until

some chip resources get saturated. NoC bandwidth is one of these common bottlenecks. The bypassing traffic optimization aims to utilize the spare NoC bandwidth more effectively and allow the reserved resources such as L1 blocks and MSHR entries to be released earlier. Consequently, PCAL can activate more bypassing threads to achieve a higher throughput.

In a memory hierarchy, when a load request misses the cache, it normally fetches the whole cache block to capture the spatial locality of the memory access streams. However, when PCAL enables extra bypassing threads, the memory requests from the bypassing threads do not allocate cache blocks, so a bypassing memory request does not need to fetch the whole cache block. Instead, it only needs to fetch the data that has been really required by the instruction. For instance, a memory request may only need to fetch a single word. However, a data cache is normally implemented with SRAM blocks, each cache block normally consists of several data segments. The size of a segment is normally bigger than a byte or a word. Our baseline simulator configuration sets the L1 data block size to be 128 bytes which are implemented as 4 32-byte segments. We optimize the PCAL mechanism by allowing the bypassing memory accesses to only fetch the corresponding segments instead of the whole cache block.

This bypassing traffic optimization can improve the throughput of PCAL in two ways. First, it reduces the NoC latency. When static PCAL or dynamic PCAL adds more threads as the bypassing threads, the bypassing traffic optimization reduces the possibility that the NoC bandwidth saturates. Conse-

quently, when the bottleneck resource is the NoC bandwidth, PCAL is able to add more bypassing threads to achieve higher throughput. Second, this optimization reduces the NoC latency. The round-trip latency of all memory requests is likely to decrease. The resources that reserved by the memory requests from cache threads, such as cache blocks and MSHR entries can be released earlier. Therefore, when the L1 block and MSHR reservation failure are major bottlenecks, this optimization can help PCAL to enable more bypassing threads.

4.6 Opportunistic Caching

We consider allowing cache blocks to be allocated *opportunistically*. That is, a non-token-holding thread can fill data into the cache if there are open cache blocks not currently used by token holders.

When cache blocks are allocated to certain number of threads, some cache blocks may not be touched by the prioritized threads during certain period. To exploit the opportunity, PCAL allows the de-prioritized threads to reserve and fill the unused cache blocks to utilize the cache more effectively. Clearing the priority information of each cache block when all prioritized threads that have accessed the block finishes, may allow more cache block be utilized by the de-prioritized threads. We evaluate several types of token clearing approaches. (1) flushing when a kernel ends, (2) flushing when all owners end, (3) flushing when the last owner ends, (4) flushing periodically. Our results show that all four methods perform similarly. Consequently, we

choose the simplest design to flush the cache block priority information when the kernel ends.

4.7 PCAL on Top of CTA Level Throttling

PCAL is orthogonal to existing thread-throttling mechanisms as it can synergistically be adopted on top of previous schemes. We conduct the experiments of PCAL on top of optimal warp-level throttling because warp is the finest granularity of execution. Prior researchers also propose to throttle the thread-level parallelism at CTA granularity. Compared to the CTA-level throttling, warp-level-throttling can tune the overall all parallelism at finer granularity to achieve better performance. Therefore, warp-throttling outperforms CTA-throttling for most benchmarks.

The implementation of PCAL on top of CTA-level throttling are very similar to implement PCAL on top of warp level throttling. Therefore the implementation details are not described again.

4.8 Results

In this section we present results for our priority-based cache allocation mechanism. As described in section 3.4, we compare to both `warp-max` and `warp-opt` in our results. We first analyze the results of static PCAL, applying each of the two optimization strategies we discuss in Section 4.3.1. We investigate the speedup, the cache miss ratio, the round-trip memory request

latency, the NoC latency and the resource reservation failure, when varying the number of Extra Bypassing Threads ($EBTN$). We discuss the reason that each application can or can not benefit from the static PCAL strategies.

Next, we explore a mechanism to dynamically select the optimization strategy and the number of extra bypassing threads to assign. We then evaluate using our bypassing traffic optimization mechanism to save the NoC bandwidth and improve the overall throughput.

4.8.1 Static Priority-based Cache Allocation

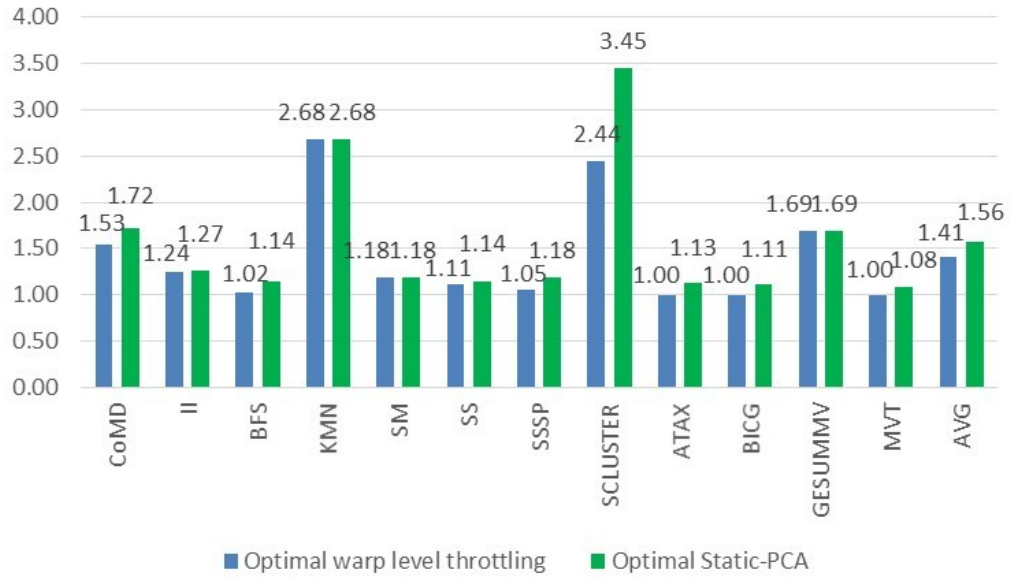
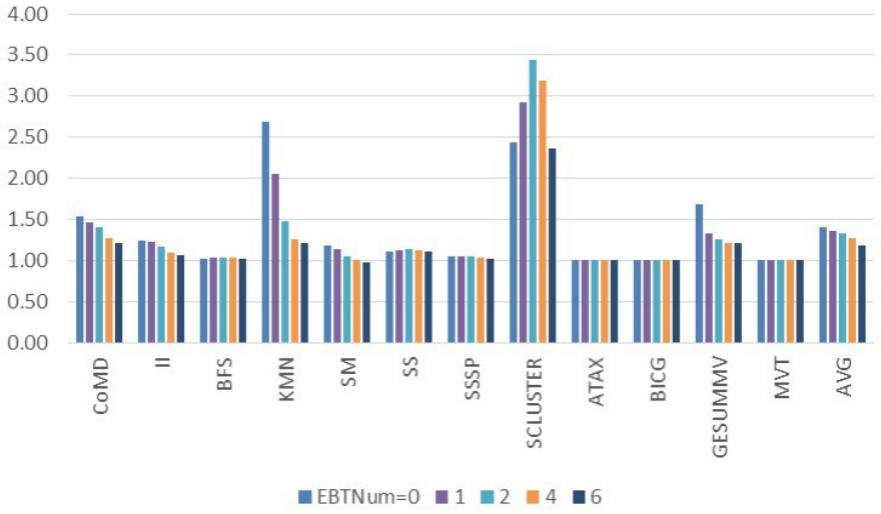


Figure 4.12: Comparing the speedup of optimal static PCAL with optimal warp-throttling. The value $EBTNum$ is selected statically to maximize the throughput.

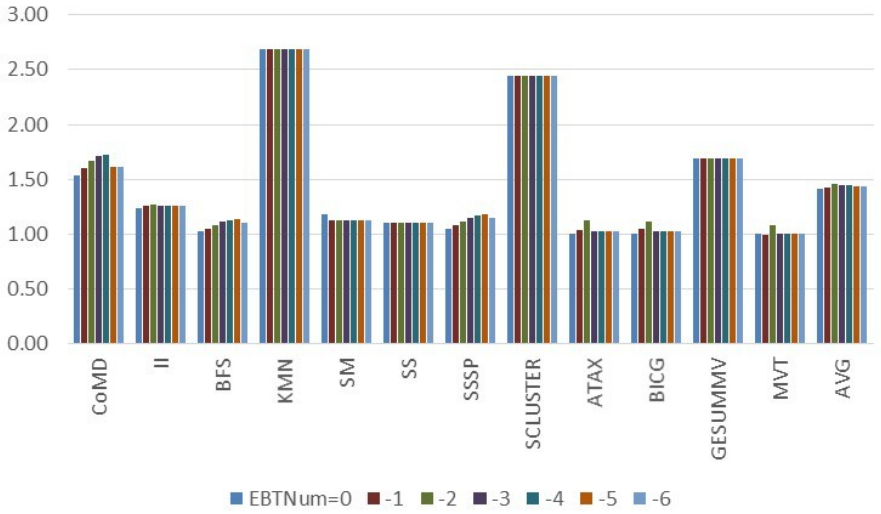
First, we evaluate the effectiveness of priority-based cache allocation at

the L1 level. Figure 4.12 compares the speedup of static PCAL with optimal $EBTNum$ value, with the speedup of optimal warp throttling `warp-opt`. The results are normalized to the original code as-is with `warp-max` warps allowed per scheduler in each SMs. We also show `warp-opt` for comparison. Among the 12 benchmarks, static PCAL enables 7 benchmarks to achieve non-negligible speedup (i.e. a speedup that is larger than 5%) over the optimal warp throttling technique. Compared to the optimal warp level throttling, static PCAL with the best $EBTNum$ value achieves an extra 15% throughput on average across all the benchmarks.

Figure 4.13 shows the detailed performance gain for our static priority-based cache allocation scheme (Static PCAL) when varying the value of $EBTNum$. Figure 4.13a and Figure 4.13b show the speedup of static PCAL when applying the optimization strategy one or two, which we discuss in Section 4.3.1. In these figures, $EBTNum = 0$ means that only warp-throttling is applied as `warp-opt`, which is the baseline of our research. Among the applications, SLUCSTER benefits from the first optimization strategy where extra bypassing threads are enabled to increase TLP while not polluting the L1 cache. CoMD, BFS, SS, SSSP, ATAX, BICG and MVT achieve higher throughput with the second optimization strategy where overall TLP is not changing while the cache hit ratio has been optimized. The other benchmarks can not benefit from any of the optimizations. We investigate the reason that each application can or can not benefit from the static PCAL strategies in rest part of this section.



(a) Speedup, when strategy one is applied to add extra bypassing threads to increase TLP



(b) Speedup, when strategy two is applied to remove threads from cache thread group to increase cache hit ratio

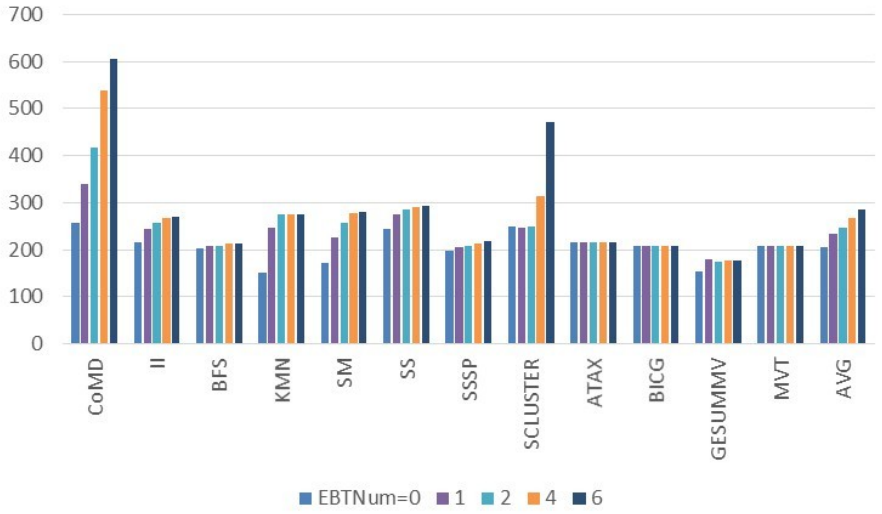
Figure 4.13: Speedup for static PCAL on L1 cache, as $EBTNum$ varies. Note $EBTNum = 0$ means that only warp-throttling is applied

4.8.1.1 Static PCAL with Strategy One: Increasing TLP While Maintaining Cache Hit Ratio

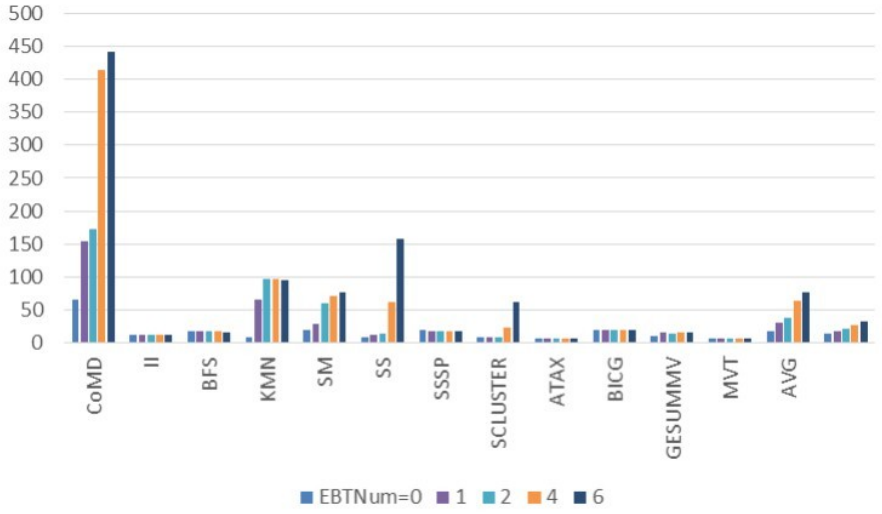
Figure 4.13a shows the speedup for static PCAL, when strategy one is applied. Extra bypassing threads are added to increase the TLP without polluting the L1 data cache. The bypassing threads do not reserve cache blocks. They do not degrade the cache hit ratio or increase the L1 block reservation failure. However they still require chip resources to fetch data blocks from the L2 cache or the main memory. The necessary resources for bypassing threads including the L1 MSHR entry, the NoC bandwidth, etc. We analyze the bottlenecks of benchmarks when optimal warp throttling is applied in Section 4.1.5. If any of the resources that a bypassing thread requires is already saturated, adding more bypassing threads leads to performance degradation instead of improvement. As shown in Figure 4.13a, among the benchmarks, SCLUSTER shows a non-negligible speedup. Other benchmarks can not benefit from the extra bypassing threads because of memory system resource saturation. The results match the bottleneck analysis results we show in Table 4.2.

Memory requests round-trip latency and NoC latency

As shown Table 4.2, CoMD, KMN and SM cannot support extra bypassing threads due to lack of NOC bandwidth. Figure 4.14 shows the round-trip latency of memory requests that fetch data from DRAM, and the average



(a) Round-Trip latency of memory requests fetching data from DRAM when static PCAL applies strategy one: adding extra bypassing threads



(b) NoC latency when static PCAL applies strategy one: adding extra bypassing threads

Figure 4.14: Round-Trip latency of memory requests fetching data from DRAM and NoC latency when static PCAL applies strategy one: adding extra bypassing threads

NoC latency when static PCAL applies strategy one to add extra bypassing threads. When the warp throttling is applied, CoMD reaches highest performance with 6 warps enabled by each scheduler. When one bypassing thread is added where $EBTNum = 1$, the NoC latency of CoMD increases from 66 to 155. The round-trip latency of CoMD increases to 339. This indicates that the NoC is highly congested. Consequently, the extra bypassing thread can not enable any extra throughput. KMN and SM experience similar NoC congestion problems as shown in the figure.

Memory pipeline stall caused by L1 MSHR reservation failure

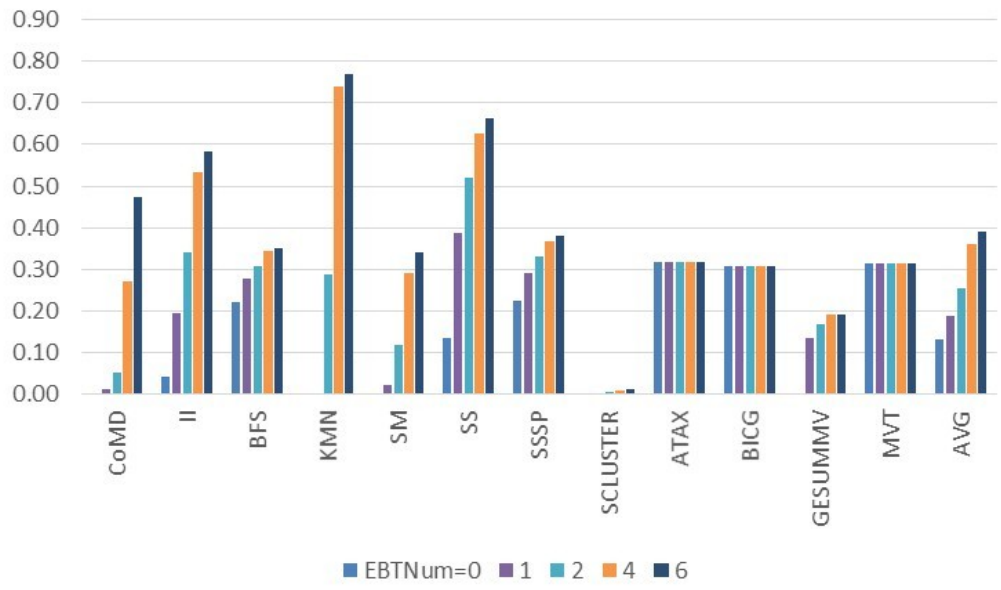


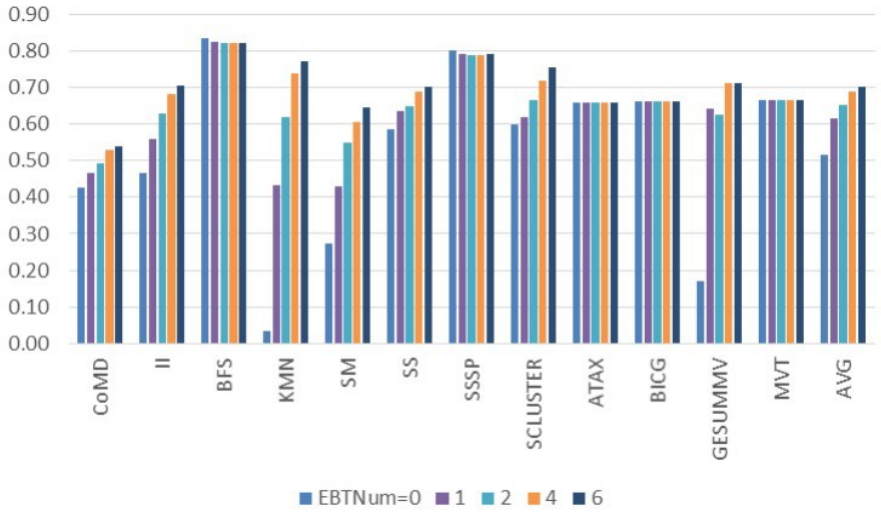
Figure 4.15: Memory Pipeline Stall caused by L1 MSHR reservation failure when static PCAL applies strategy one: adding extra bypassing threads

Figure 4.15 shows the average Memory Pipeline Stall caused by L1 MSHR reservation failure when static PCAL applies strategy one to add extra bypassing threads. In the current PCAL implementation, the memory requests from the bypassing threads reserve the MSHR entries along with the token-holder threads. Our baseline GPU configuration, as shown in Table 3.4, integrates a MSHR table with only 32 entries in each SM. As shown Table 4.2, when optimal throttling is applied without adding more bypassing threads, II, BFS, SS and SSSP are already bottlenecked by the L1 MSHR reservation problem. The extra bypassing threads cause significant MSHR saturation except in SCLUSTER. Enlarging MSHR table capacity might enable PCAL to achieve higher performance on more benchmarks. We leave this study for future work.

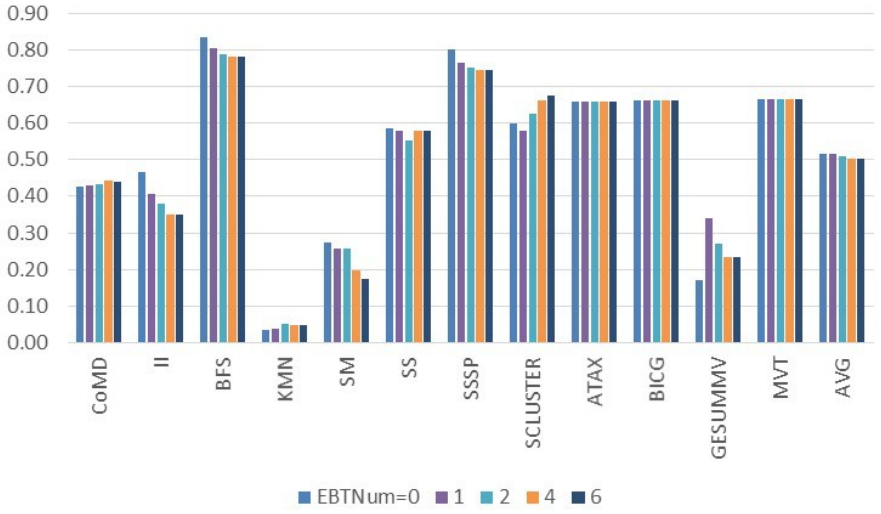
Overall, cache threads and bypassing threads L1 cache miss ratio

With optimization strategy one, static PCAL adds extra bypassing threads to increase the TLP. We expect the cache threads (token holder threads) to maintain the cache hit ratio as if only optimal warp throttling is applied.

Figure 4.16 shows the overall L1 cache miss ratio and the cache miss ratio of the cache threads (token holder) when static PCAL applies strategy one to add extra bypassing threads. As shown in Figure 4.16b, when static PCAL enables extra bypassing threads, the cache hit ratio of the cache threads



(a) Overall L1 cache miss ratio when static PCAL applies strategy one: adding extra bypassing threads



(b) L1 cache miss ratio of the cache threads (token holder) when static PCAL applies strategy one: adding extra bypassing threads

Figure 4.16: Overall L1 cache miss ratio and the cache miss ratio of the cache threads (token holder) when static PCAL applies strategy one: adding extra bypassing threads

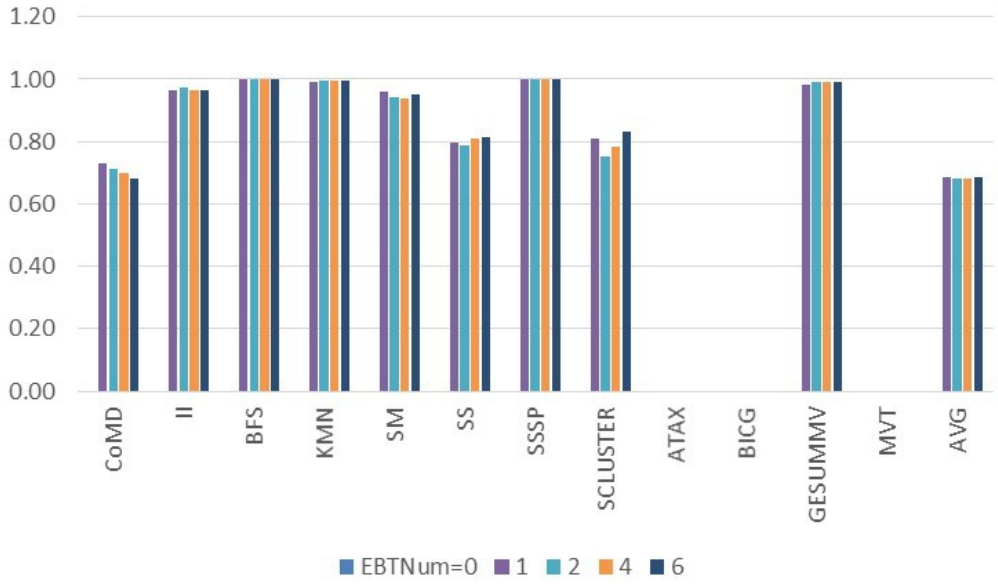


Figure 4.17: L1 cache miss ratio of the bypassing threads (non-token holder) when static PCAL applies strategy one: adding extra bypassing threads

does not increase except in GESUMMV. GESUMMV is not bottlenecked by most chip resources. We expect GESUMMV to benefit from adding extra bypassing threads. However, when one extra bypassing thread is added, the L1 hit ratio of the cache threads increases from 17% to 34% which is not as we expect. The reason is that the opportunistic caching optimization allows the non-token holder to reserve cache blocks when the block is not occupied by token holders.

The current PCAL implementation allows the bypassing threads to probe the L1 cache so that the bypassing threads have the possibility to hit the cache when they share data with the cache threads. Figure 4.17 shows the L1 cache miss ratio of the bypassing threads when static PCAL enables extra bypassing threads. As shown in this figure, ATAX, BICG and MVT achieves their highest performance with their highest TLP, there is no bypassing thread that can be enabled. Consequently, their bypassing thread L1 hit ratio is zero. The bypassing threads from CoMD, SS and SCLUSTER show non-negligible cache hits. Other benchmarks show a nearly 100% cache miss ratio which means that allowing bypassing threads does not increase their cache hit ratio.

Parallel Starvation

As shown in Table 4.1 and Table 4.2, ATAX, BICG and MVT achieve their highest performance at the maximum number of threads allowed. They experience the parallel starvation problem thus there is no extra bypassing

thread can be added even the scheduler allows a larger maximum number of threads. Therefore, these four benchmarks can not benefit from extra bypassing threads.

4.8.1.2 Static PCAL with Strategy Two: Increasing Cache Hit Ratio While Maintaining TLP

As we discuss in Section 4.8.1.1, when the spare chip resources are not sufficient to support extra bypassing threads, static PCAL can not enable bypassing threads. In this case, optimization strategy two is applied. Some of the cache threads are changed to bypassing threads to avoid thrashing the cache while the total TLP is maintained.

Figure 4.13b shows the speedup for static PCAL, when optimization strategy two is applied. When static PCAL reduces the number of the cache threads, CoMD, BFS, SSSP, ATAX, BICG and MVT exhibit significant throughput improvement. The other applications do not get benefits from this optimization. We analyze them in the rest of this section.

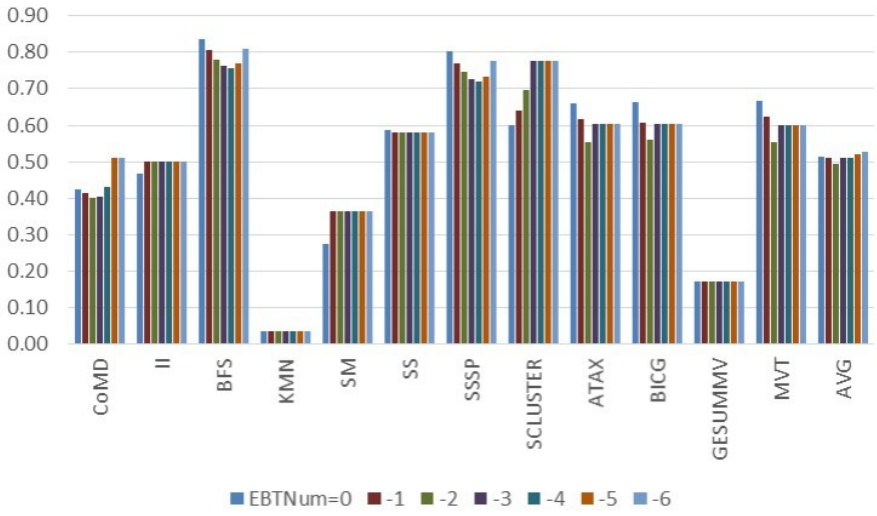
Benchmarks with parallel starvation problem, such as ATAX, BICG and MVT etc., can not benefit from optimization strategy one, because no extra bypassing threads can be added. For these benchmarks, static PCAL can apply strategy two to reduce the cache miss ratio to improve the performance. As shown in Figure 4.13b, ATAX, BICG and MVT achieve higher performance when static PCAL applies strategy two.

Overall, cache threads and bypassing threads L1 cache miss ratio

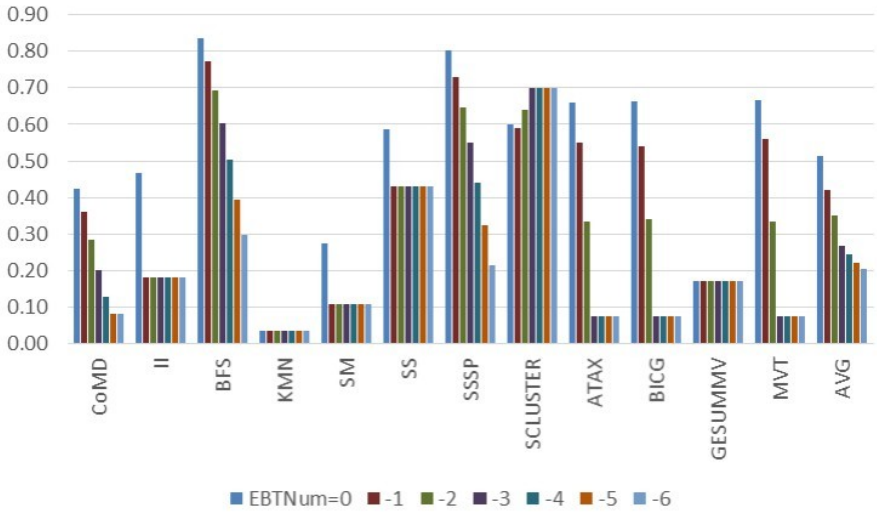
With optimization strategy two, static PCAL reduces the number of cache threads to improve the cache hit ratio and maintain the overall TLP. We expect the working set of the cache threads (token holder threads) to reside in the cache to minimize their cache misses. The bypassing threads have to experience more cache misses. However, when an application experiences cache thrashing, keeping a fraction of data resident in cache is expected to improve the overall cache hit ratio.

Figure 4.18b shows the L1 miss ratio of the cache threads (token holder) when static PCAL applies strategy two to reduce the number of cache threads to improve the cache hit ratio. Most benchmarks, except KMN, GESUMMV and SCLUSTER, exhibit much lower L1 miss ratios for the cache threads when the number of cache threads is reduced. Our baseline configuration is the optimal warp level throttling. As Shown in Table 4.1, the `warp-opt` value of KMN and GESUMMV is one. The current PCAL implementation maintains at least one thread to feed the cache. Therefore, for KMN and GESUMMV, the actual number of cache threads does not change even when the value of $EBTNum = 0$ changes from -1 to -6. There are no bypassing threads enabled for these two benchmarks when static PCAL applies optimization strategy two.

SCLUSTER exhibits a different trend in the cache threads L1 miss ratio. Unlike the other benchmarks, its L1 miss ratio increases when the



(a) Overall L1 cache miss ratio when static PCAL applies strategy two: reducing the number of the cache threads to improve the cache hit ratio.



(b) L1 cache miss ratio of the cache threads (token holder) when static PCAL applies strategy two: reducing the number of the cache threads to improve the cache hit ratio

Figure 4.18: Overall L1 cache miss ratio and the cache miss ratio of the cache threads (token holder) when static PCAL applies strategy two: reducing the number of the cache threads to improve the cache hit ratio

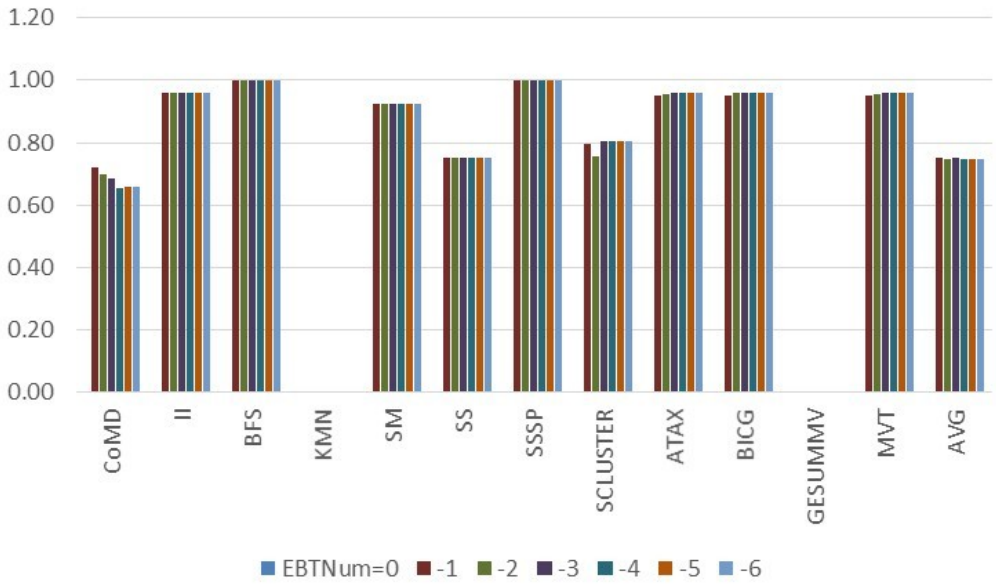


Figure 4.19: L1 cache miss ratio of the bypassing threads (non-token holder) when static PCAL applies strategy two: reducing the number of the cache threads to improve the cache hit ratio

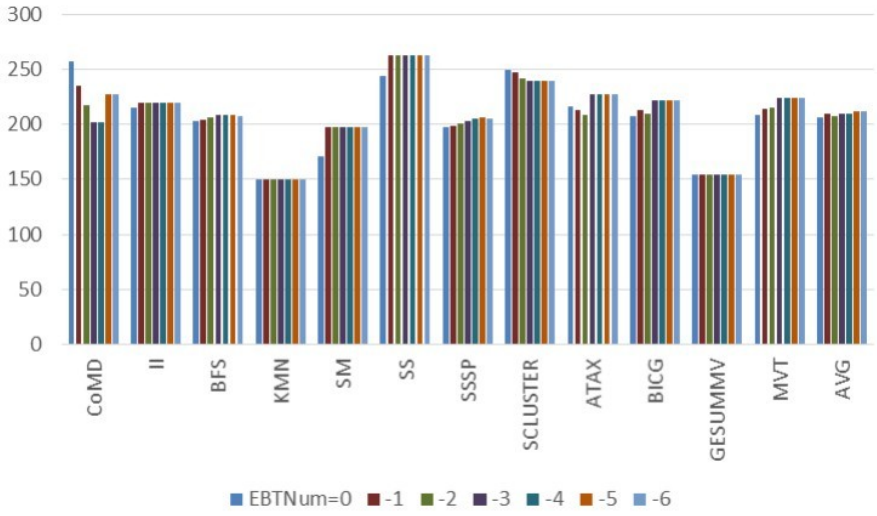
number of cache threads is reduced. The reason is that SCLUSTER has a large amount of inter-warp data locality. Reducing the number of cache threads degrades the opportunity to capture inter-warp locality and thus leads to a higher miss ratio.

Figure 4.18a shows the overall L1 cache miss ratio when static PCAL applies strategy two to reduce the number of cache threads to improve the cache hit ratio. As shown in this figure, several applications show a bath-tub-shaped L1 miss ratio function as the number of cache threads is reduced. The overall L1 miss ratio is a mixed effect of the cache threads' miss ratio reduction and the bypassing threads' cache miss increase. This shows that the overall cache hit ratio is sensitive to the $EBTNum$ value.

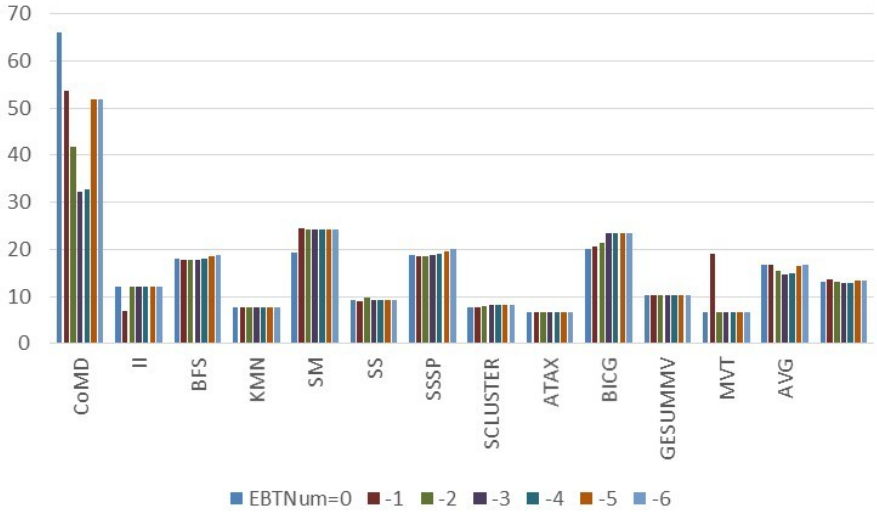
Round-Trip Latency and NoC Latency

Static PCAL applies strategy two to reduce the number of the cache threads and improve cache hit ratio. At the same time, the total TLP does not increase. Consequently, when optimization strategy two is enabled, static PCAL reduces the number of outstanding memory requests and thus alleviates the chip resource contention problem.

Figure 4.20a shows the round-trip latency of the memory requests that fetch data from DRAM, and the average NoC latency when static PCAL applies strategy two to reduce the L1 miss ratio. The applications for which this optimization strategy enables lower L1 miss ratio, such as CoMD and SSS etc. show lower round-trip latency as we expect. The other benchmarks that



(a) Round-Trip latency of the memory requests fetching data from DRAM when static PCAL applies strategy two



(b) NoC latency when static PCAL applies strategy two

Figure 4.20: Round-Trip latency of the memory requests fetching data from DRAM and NoC latency when static PCAL applies strategy two: reducing the number of the cache threads to improve the cache hit ratio

do not achieve higher throughput with this optimization, such as SCLUSTER and SS etc. do not show noticeable round-trip latency increase. Figure 4.20b shows the NoC latency when static PCAL applies strategy two. It mostly exhibits a similar trend as the round-trip latency does.

Memory pipeline stall Ratio

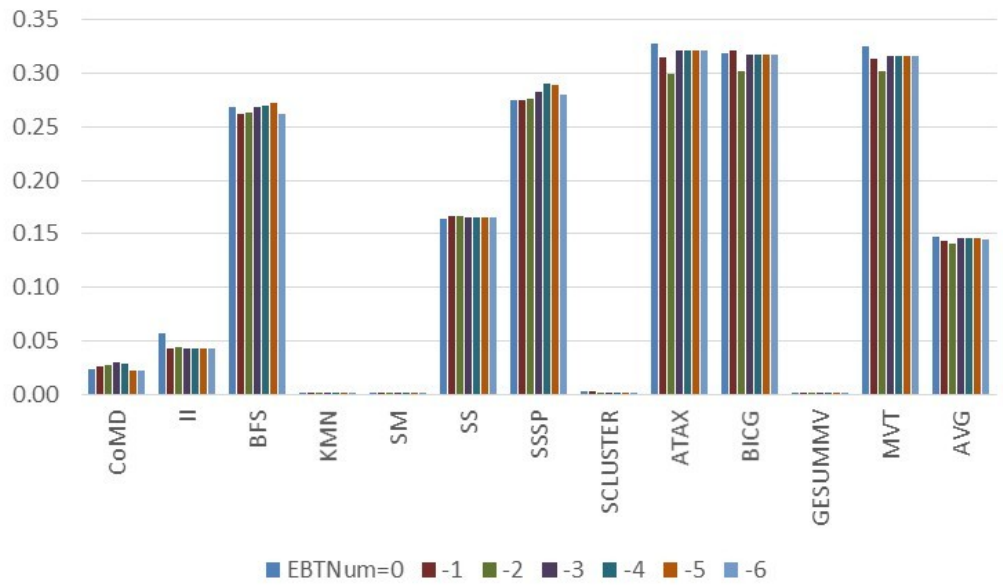


Figure 4.21: Ratio Memory Pipeline Stall when static PCAL applies strategy two:reducing the number of the cache threads to improve the cache hit ratio

Using strategy two to reduce the cache miss ratio, static PCAL can reduce the number of outstanding memory requests and is expected to experience less chip resource saturation. Figure 4.21 shows the average memory

pipeline stall ratio of static PCAL when applying strategy two. All benchmarks experience smaller or the same ratio of memory pipeline stalls as expected.

4.8.2 Dynamic Priority-based Cache Allocation

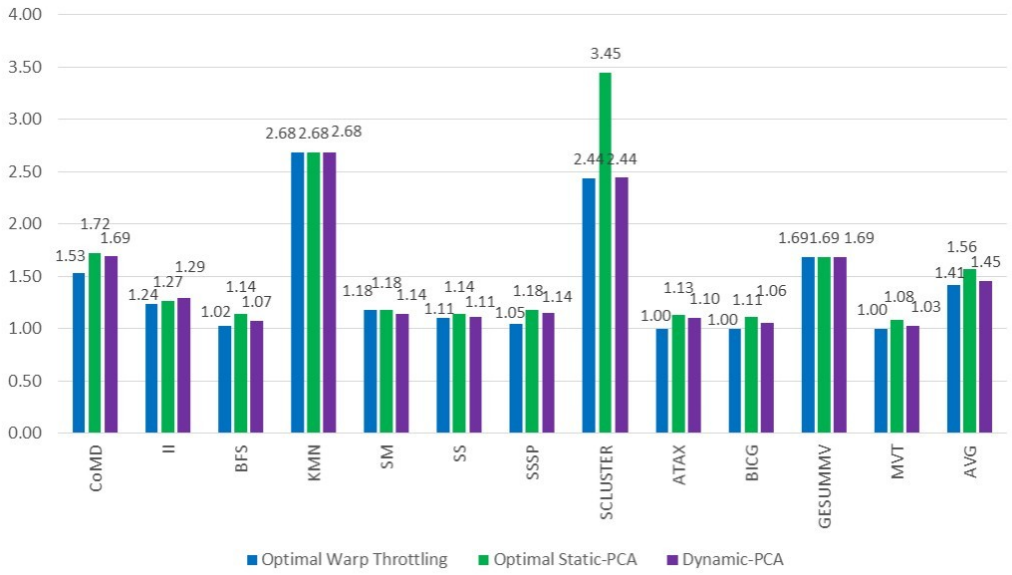


Figure 4.22: Comparing the speedup of the best static PCAL and dynamic PCAL:(1) optimal Warp Throttling (2) optimal static PCAL (3) dynamic PCAL. Normalized to baseline which allows max warp/scheduler

To evaluate the performance of dynamic PCAL, we compare the speedup for the following mechanisms, all results have been normalized to the baseline.

- **Baseline**. All schedulers launch the maximum number of warps that the on-chip resources allow (`warp-max`).
- **Optimal Warp Throttling**: Throttling with the optimal warp number (`warp-opt`). All schedulers launch the optimal number of warps that

results in the highest throughput.

- **Optimal static PCAL**: Bypassing-L1 with the optimal warp number and optimal token number. All schedulers launch the optimal number of warps and apply the bypassing-L1 scheme with the optimal number of tokens

- **Dynamic PCAL**: Bypassing-L1 with Dynamic token allocation. Applications have been throttled by warp, `warp-opt` is known as an input. The dynamic token allocation algorithm detects and configures the number of extra bypassing threads ($EBTNum$) that the bypassing-L1 scheme can apply. Rogers, et al. [77] propose a dynamic warp throttling scheme. We approximate the dynamic warp throttling mechanism with the optimal warp number and show that the dynamic token allocation algorithm over a warp throttling scheme can achieve almost the same performance improvement that the scheme with optimal warp and optimal $EBTNum$ can reach.

Figure 4.22 compares the speedup of the 3 mechanisms. As shown in the figure, for most of the applications, the speedups of **Static PCAL** and **Dynamic PCAL** are very close. This figure demonstrates that the dynamic PCAL scheme is able to achieve the majority of the performance improvements that the best static PCAL can achieve for most applications.

SCLUSTER is the exception. When dynamic PCAL is enabled, it does not achieve any speedup beyond optimal warp throttling. The reason is

two-fold. (1) the throughput of SCLUSTER is very sensitive to the number of the extra bypassing threads PCAL enables. (2) SCLUSTER consists of many unstable program phases so that dynamic PCAL can not determine the optimal value of $EBTNum$.

As we discuss in Section 4.8.1.1, static PCAL applies optimization strategy one for SCLUSTER to add the minimum number of extra bypassing threads that saturates chip resources. As shown in Figure 4.13a, the optimal value of $EBTNum$ for SCLUSTER is 4. However, the speedup of SCLUSTER is very sensitive to the number of extra bypassing threads. When $EBTNum$ increases from 4 to 6, its speedup drops significantly from 345% to 237%, which is lower than the speedup of the optimal warp throttling.

As we discuss in Section 4.8.1.2, Dynamic PCAL applies a Hill-Climbing algorithm to decide the best value of $EBTNum$. However, SCLUSTER consists of unstable program phases, so dynamic PCAL can not find the best value of $EBTNum$ before new phase is detected. As a result, dynamic PCAL fails to improve the performance of SCLUSTER.

4.8.3 Bypassing Traffic Optimization

As we discussed in Section 4.5, we optimize the PCAL mechanism by allowing the bypassing memory accesses to only fetch the corresponding data segments they need instead of the whole cache blocks. The bypassing traffic optimization (BTO) is able to utilize the spare NoC bandwidth more effectively. This optimization helps PCAL in two ways. (1) more threads can be

added as the bypassing threads and thus the system achieves a higher throughput. (2) it allows the reserved resources such as L1 blocks and MSHR entries t .

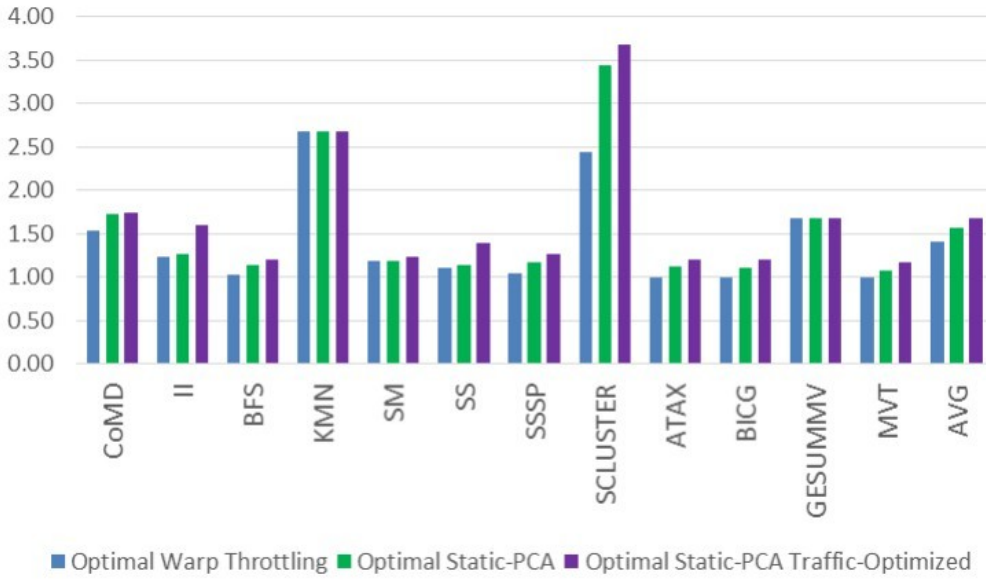


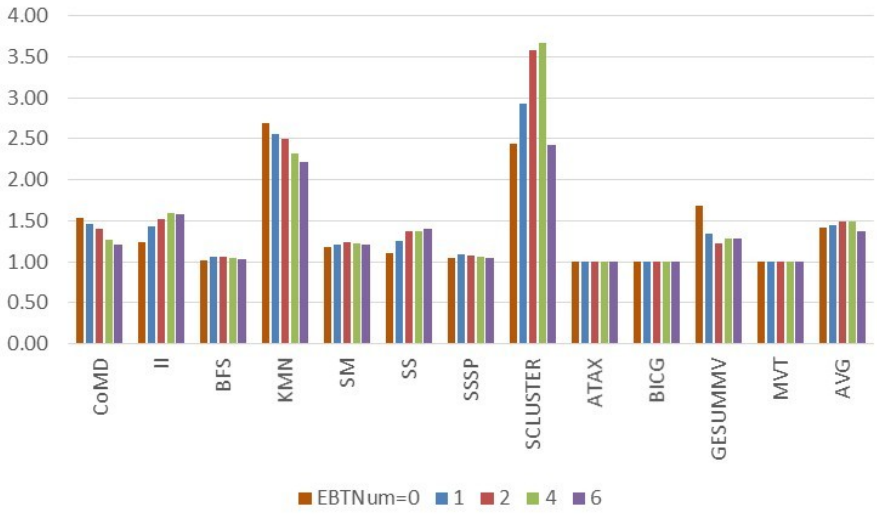
Figure 4.23: Comparing the speedup of optimal warp level throttling, optimal static PCAL and static PCAL with with NoC traffic optimization (Static PCAL-BTO).(Normalized to baseline which allows max warp/SM)

Figure 4.23 compares the speedup of three mechanisms: (1) warp throttling with optimal warp number per scheduler (`warp-opt`), (2) static PCAL with optimal $EBTNum$ value, (3) static PCAL with optimal $EBTNum$ value. Bypassing traffic optimization enabled (Static PCAL-BTO). All the results are normalized to the original code as-is with `warp-max` warps allowed per scheduler in each SM.

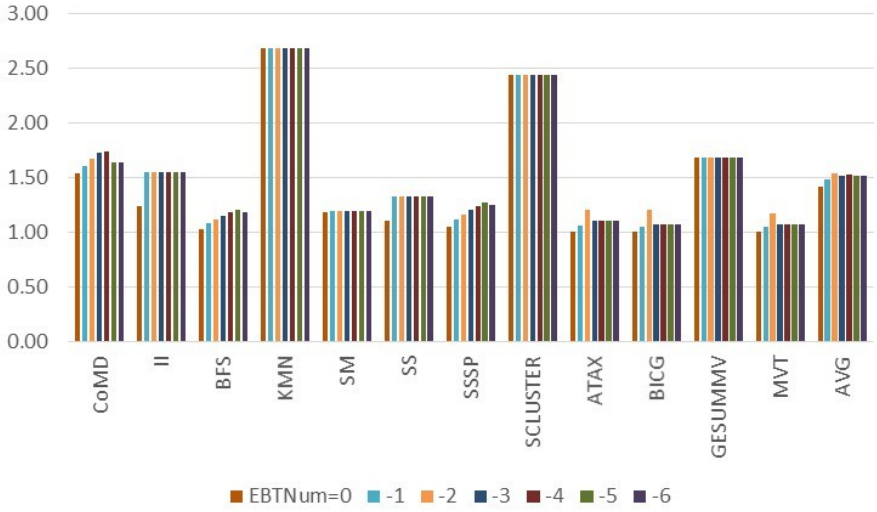
Compared to the baseline which allows the maximum warp count per SM, warp level throttling, static PCAL and static PCAL-BTO achieve average

speedups of 41%, 56% and 67% respectively across the 12 applications. The bypassing traffic optimization improves the performance of static PCAL significantly. As shown in the figure, when the NoC traffic is optimized, 10 out of the 12 benchmarks can benefit from the PCAL technique. KMN and GESUMMV, unlike the other benchmarks, still can not benefit from PCAL even when the NoC traffic has been optimized. The reason is that, when PCAL applies strategy one, even the bypassing threads only fetch the data segments that the memory instructions require instead of the whole cache line, the spare chip resources are still not sufficient to support an extra bypassing thread. When PCAL applies strategy two, the optimal warp number per scheduler of both these two benchmarks is one. As we indicated previously, PCAL keeps at least one thread as a cache thread. Consequently, neither strategy one nor strategy two of PCAL can help KMN and GESUMMV to achieve extra throughput beyond pure warp level throttling.

Among the 12 benchmarks, static PCAL enables 7 benchmarks to achieve non-negligible speedup (i.e. a speedup that is larger than 5%) beyond optimal warp level throttling. II, SM, SS, KMN and GESUMMV can not achieve higher throughput by the static PCAL mechanism. Static PCAL-BTO, which enables the bypassing traffic optimization, not only improves the performance of the 7 benchmarks that benefit from static PCAL but also improve the performance of SS, SM and II. We investigate the performance effect of the bypassing traffic optimization across our benchmarks in the rest of this section.



(a) Speedup, when strategy one is applied to add extra bypassing threads to increase TLP



(b) Speedup, when strategy two is applied to remove threads from cache thread group to increase cache hit ratio

Figure 4.24: Speedup for static PCAL with bypassing traffic optimization, as $EBTNum$ varies. Note $EBTNum = 0$ means that only warp-throttling is applied

Figure 4.24a and Figure 4.24b show the speedup of static PCAL-BTO when applying the two optimization strategies discussed, which we discuss in Section 4.3.1. When strategy one is applied, PCAL allows more threads to be enabled as bypassing threads. The bypassing traffic optimization minimizes the amount of bypassing traffic thus it can allow more bypassing threads to achieve higher throughput. For instance, without the bypassing traffic optimization, static PCAL can enable two bypassing threads for SCLUSTER (as shown in Figure 4.13a), which leads to a speedup of 3.45X for SCLUSTER. When the bypassing traffic optimization is enabled, as shown in Figure 4.24a, four bypassing threads can be enabled without degrading performance. It leads to a speedup of 3.67X for SCLUSTER. Another example is SM, where static PCAL can not improve its performance beyond the speedup of 18% that the warp level throttling enables, because the spare resources are not sufficient for even one bypassing thread. Static PCAL-BTO with the bypassing traffic optimization enabled, in contrast, enables two bypassing threads and achieves a speedup of 23% for SM. Similarly, static PCAL-BTO enables SS and II to achieve a higher throughput by allowing more bypassing threads.

When strategy two is applied, PCAL reduces the number of cache threads to improve the cache hit ratio while maintaining the overall TLP. The benchmarks that benefit from strategy two are in general chip resource bound, which makes adding more threads as with strategy one not applicable. The bypassing traffic optimization minimizes the amount of bypassing traffic thus the NoC latency and the overall round-trip latency decrease. As a result,

these benchmarks become less bounded thus achieve higher throughput.

For instance, for BFS, static PCAL without the bypassing traffic optimization applies strategy two, reduces the number of cache threads from 8 to 3, decreases the overall L1 miss ratio from 83% to 76% and achieves a speedup of 13%. As a contrast, optimal warp throttling only enables a speed up of 2.2% (as shown in Figure 4.13b). When the bypassing traffic optimization is enabled, as shown in Figure 4.24b, Static PCAL-BTO maintains a L1 miss ratio of 77% similar to that of static PCAL. However, compared to static PCAL, static PCAL-BTO reduces the NoC latency from 18 to 7 and reduces the average round-trip latency from 209 to 193. As a result, we obtain a speedup of 20% for BFS. Similarly, static PCAL-BTO enables SSSP, CoMD, ATAX, BICG and MVT to achieve higher throughput by reducing the NoC latency and the round-trip access latency.

4.8.4 Applying PCAL on both L1 and L2

Figure 4.26 compares the speedup of the optimal static PCAL on L1 and optimal static PCAL on L1&L2. Compared to static PCAL on L1, static PCAL on L1&L2 gets a lower throughput for eight benchmarks. Among them, five benchmarks' throughput decrease more than 5%. BFS, SSSP, ATAX, BICG, and MVT experience performance degradation of 10%, 10%, 9%,8% and 7% respectively. For the other benchmarks, static PCAL on L1&L2 enables similar speedup as static PCAL on L1 does.

The reason that static PCAL on L1&L2 leads to a significant perfor-

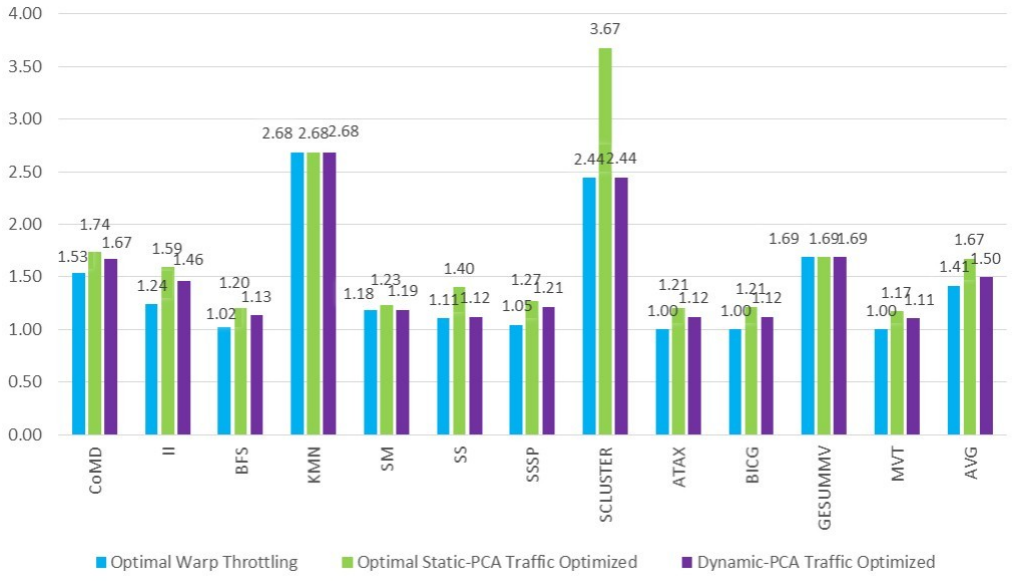


Figure 4.25: Comparing the speedup of the best static PCAL and dynamic PCAL both with bypassing traffic optimization:(1) optimal Warp Throttling (2) optimal static PCAL (3) dynamic PCAL. Normalized to baseline which allows max warp/scheduler

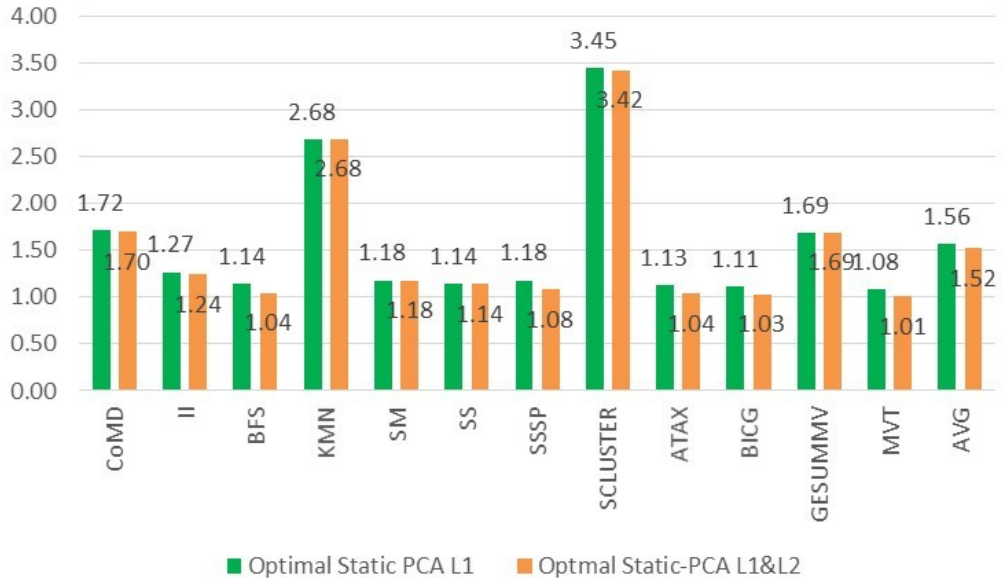
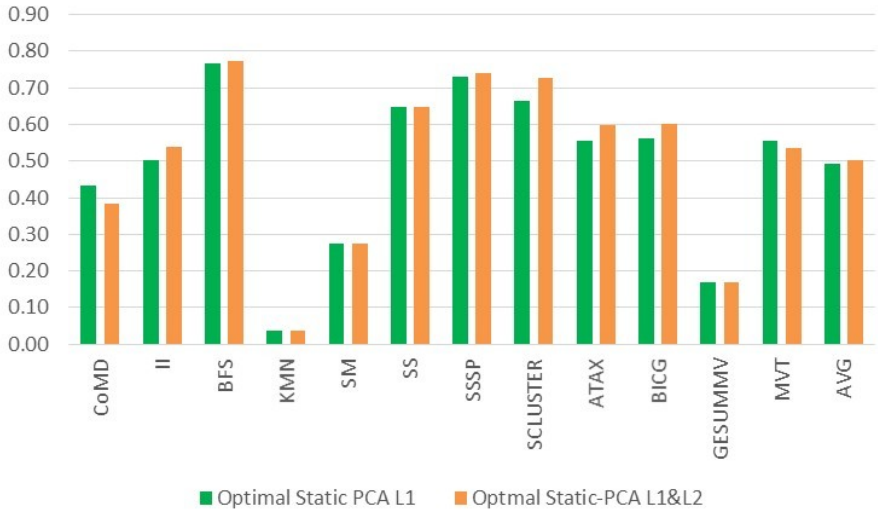
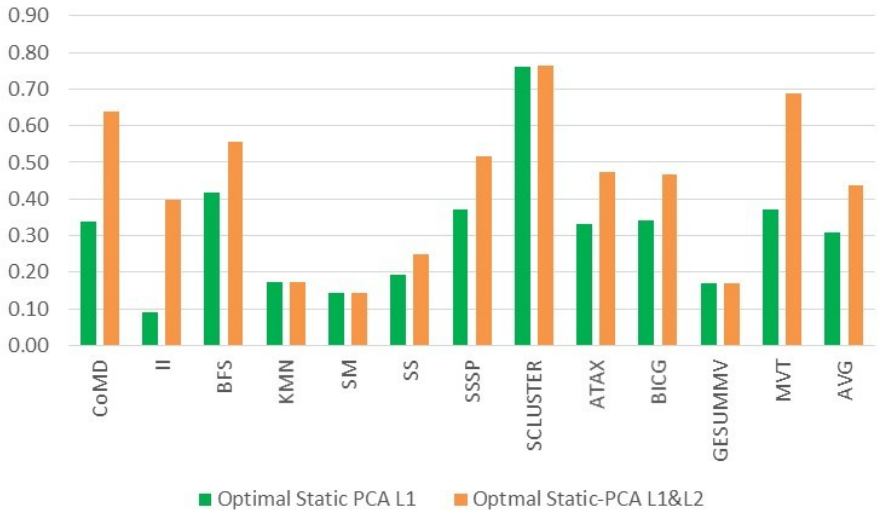


Figure 4.26: Comparing the speedup of the best static PCAL on L1 and static PCAL on L1&L2. Normalized to baseline which allows max warp/scheduler

mance degradation is that static PCAL on L1&L2 forces the bypassing threads to not only bypass the L1 cache but also the L2 cache. If the L2 cache is not saturated, bypassing L2 leads to a higher L2 miss ratio. Figure 4.27 compares the L1 and L2 miss rate of optimal static PCAL on L1 and optimal static PCAL on L1&L2. The L1 miss rate between these two schemes are very close as we expect. Figure 4.27b compares the L2 miss rate of these two schemes. The L2 miss ratio of static PCAL on L1&L2 scheme is 44%. As a contrast, the L2 miss ratio of static PCAL on L1 is 31%. Among these benchmarks, KMN and GESUMMV achieve their highest performance when there is no bypassing thread. Therefore, static PCAL on L1&L2 does not degrade the L2 miss ratio of these two benchmarks.



(a) Comparing L1 miss rate of the best static PCAL on L1 and static PCAL on L1&L2.



(b) Comparing L2 miss rate of the best static PCAL on L1 and static PCAL on L1&L2.

Figure 4.27: Comparing L1 and L2 miss rate of the best static PCAL on L1 and static PCAL on L1&L2.

Allowing bypassing threads to bypass L2 might be helpful when the L2 cache is already saturated by the cache threads. Therefore, there might be a further performance opportunity available if the decision can be made adaptively. We leave the adaptive L2 bypassing as future work.

4.8.5 Results Summary

In this section, we have evaluated the static PCAL, dynamic PCAL, bypassing traffic optimization and static PCAL on L1&L2. As we show, compared to PCAL on L1, PCAL on L1& L2 leads to performance degradation for most of the benchmarks. Therefore we only summarize the speedup of other schemes here.

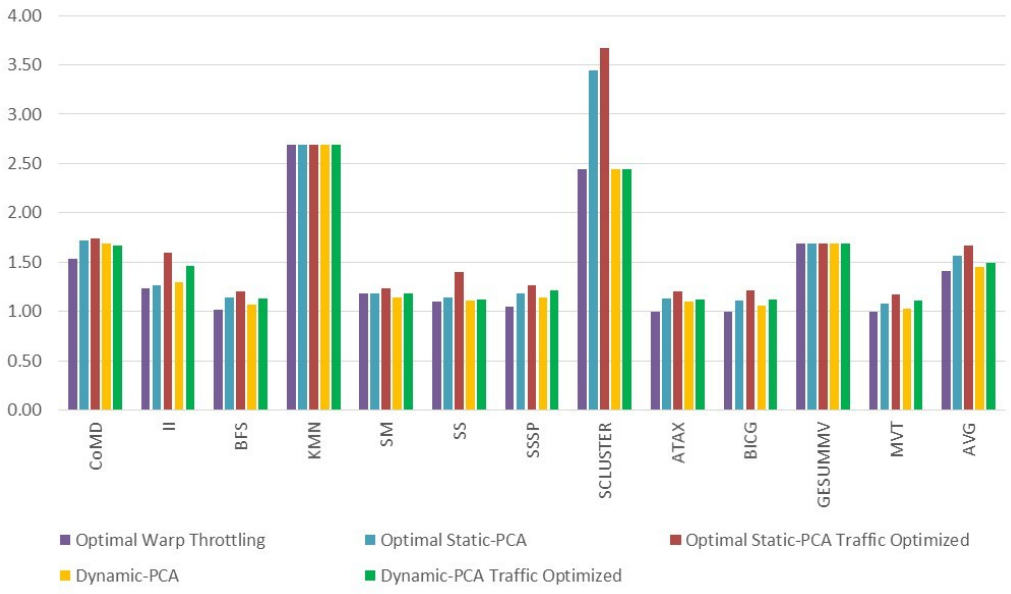


Figure 4.28: Overall speedup (normalized to baseline with maximum warp per scheduler)

Figure 4.28 summarizes our best performance for a variety of configurations on each application. For each configuration, we show results for the best performing warp throttling, optimal static PCAL, optimal static PCAL with bypassing traffic optimization, Dynamic PCAL and dynamic PCAL with bypassing traffic optimization. Overall, we observe substantial improvements vs. off-the shelf applications, especially when the bypassing traffic optimization is applied. The dynamic PCAL scheme is able to achieve the majority part of the performance improvements that the best static PCAL can achieve for most applications. While warp-throttling can provide benefit over the original applications, our approach provides additional improvement. Overall, for these applications, we observe 56% improvement over the original as-is code, a 16% improvement over a better-tuned warp-throttling baseline.

4.9 Conclusion

While massively threaded processors such as GPUs are able to provide high throughput, implementing a high throughput memory system becomes very challenging for two reasons. First, the gap between the peak arithmetic capability and the off-chip bandwidth will inevitably be larger. Second, exploiting locality in these systems can be difficult because of the competition for cache capacity by the threads.

In this chapter, we demonstrate that the memory system needs to separate the concerns of cache thrashing and chip resources saturation. We observe two opportunities to utilize the memory system more effectively. First, when

chip resources, such as off-chip bandwidth or NoC bandwidth etc., get saturated, we show that cache-sensitive workloads have the opportunity to increase cache locality, but not merely by increasing cache size. Reducing the threads that compete for the cache and maintaining the total TLP by allowing other threads to bypass the cache are the key elements to enable better usage of the cache resources without sacrificing overall parallelism. Second, when the total TLP has to be compromised to maintain a decent cache hit-ratio to maximize the overall throughput, we demonstrate that adding a minimum number of bypassing threads to utilize the spare resources can effectively improve the throughput.

To exploit these two performance opportunities, we propose and evaluate a priority-based cache allocation (PCAL) mechanism that gives preferential access to the cache and other on-chip resources to a subset of the threads and allows the remaining threads to bypass the caches. Our priority-based approach reduces cache contention and employs effectively the chip resources that would otherwise go unused by a pure thread throttling approach.

Our results show that PCAL enhances the capabilities of thread throttling, enabling an increase in performance of 18% over the optimal warp throttling scheme on cache-sensitive workloads. Our research also shows that a dynamic algorithm that automatically selects the number of bypassing threads to adapt to the locality profile of each application is competitive with the best off-line static bypassing threads count selection. This result is significant as it shows that locality-based performance tuning need not require expert

programming skills. Our results also show that cache insensitive applications are not hurt by PCAL, as the algorithms easily default to all threads having the same priority. We also expect PCAL to show benefits on emerging GPU workloads that are less regular in their inter-thread data access patterns and that have working sets that can fit in the on-chip caches.

Chapter 5

Thrashing-Resistant GPU L1 Cache Replacement and Bypassing algorithms

In this chapter, we propose AgeLRU and Dynamic-AgeLRU, new GPU cache replacement and bypassing algorithms that adapt to GPU thread scheduling algorithms to alleviate GPU L1 cache thrashing and thus improve the GPU memory system throughput.

First, we motivate this research by investigating the feasibility of applying a set of thrashing-resistant CPU cache algorithms to the GPU cache hierarchy, including the Bimodal Insertion Policy (BIP) mechanism [71] and the Dynamic Insertion Policy (DIP) mechanism [71].

Next, we propose the AgeLRU and Dynamic-AgeLRU mechanisms, which are thread scheduling aware replacement and bypassing algorithms to overcome the thrashing problem. When selecting a collection of memory blocks to reside in the cache, AgeLRU minimizes the number of warps that share the cache-resident blocks by prioritizing older warps. At replacement, it considers not only the last reference time but also the age of the warp fetching the block. Dynamic-AgeLRU selects the AgeLRU or the LRU algorithm adaptively based on parallel voting mechanism.

Next, we introduce mechanisms for implementing the AgeLRU and Dynamic-AgeLRU algorithms. We describe the initial implementation of these algorithms as both replacement and bypassing algorithms. The mechanism to detect the blocks fetched by inactive warps is discussed. We introduce the parallel voting mechanism to support Dynamic-AgeLRU.

Finally, we evaluate the new replacement and bypassing policies with a set of cache-sensitive benchmarks. The results are compared to not only the baseline but also to the adapted BIP/DIP mechanisms. Compared to the LRU algorithm, the AgeLRU replacement, bypassing and bypassing with traffic optimization algorithms enable increases in performance of 4%, 8% and 28% respectively across fourteen cache-sensitive benchmarks. Our results show that Dynamic-AgeLRU algorithms can avoid degrading the performance of non-thrashing applications by selecting the LRU algorithm, while achieving the majority of the performance improvements that the AgeLRU algorithms can achieve for most cache sensitive applications.

5.1 Motivation

In this section, we motivate our research by analyzing the problems and the improvement opportunities of applying thrashing-resistant CPU cache algorithms to a GPU L1 cache. We demonstrate that applying thrashing-resistant CPU cache algorithms, such as Bimodal Insertion Policy (BIP) and Dynamic Insertion Policy (DIP) etc.to the GPU cache can not effectively address GPU L1 cache thrashing. At the end, we discuss opportunities to address

the GPU thrashing problem beyond BIP/DIP. We observe that the cache replacement and bypassing algorithms can adapt to the scheduling algorithm to increase the overall throughput.

Cache Thrashing Problem

As we demonstrate in Section 1.1, the GPU per-thread cache capacity is extremely limited. As a result, keeping the working set of all threads resident in the primary cache is infeasible on a fully-occupied SM.

The most common replacement algorithms, such as the Least Recently Used (LRU) algorithm and its approximation Not Recently Used (NRU) [27] [87], assume that the most recently referenced block is referenced immediately. This assumption does not hold when an application experiences a cache thrashing problem. In this case, although many cache blocks will be reused in the future, the new fetched blocks push them from the MRU state to the LRU state. The blocks then get evicted before the reuse can be captured. For GPU applications, the massive number of hardware threads that a GPU supports often leads to the cache thrashing problem.

The way to solve the thrashing problem is to only allow a small fraction of the blocks to be resident in the cache until they get referenced. The other cache blocks are either put in the LRU state or are forced to bypass the cache to avoid evicting the other cache blocks.

Thrashing-resistant CPU cache algorithms

Cache thrashing on a CPU cache has been a well-known problem and has been widely studied. Researchers have proposed thrashing-resistant CPU cache management algorithms. For instance, the Bimodal Insertion Policy (BIP) [71] and its variants [31] [97] address the cache thrashing problem by randomly selecting a small fraction of all cache blocks to reside in the MRU state or the non-LRU state while the majority of cache blocks are kept in the LRU state. The BIP algorithm effectively alleviates the cache thrashing problem when the working set of an application is much bigger than the cache capacity. To avoid degrading the performance of no-thrashing workloads, a Dynamic Insertion Policy (DIP) [71] is proposed to select between the BIP and LRU algorithms adaptively.

We adapt both BIP and DIP to the GPU L1 cache. The results are introduced and analyzed in the results section. We observe that these techniques can improve the GPU cache and overall performance for some applications. However, there are two problems that limit the feasibility of adapting these CPU cache algorithms to the GPU L1 cache. (1) BIP randomly selects memory blocks to reside in the cache. It tends to activate all warps concurrently and thus it increases the total working set size. (2) DIP relies on the set-dueling mechanism to evaluate two algorithms on different sets of the same cache. In a GPU, the thread interleaving and the cache access pattern may change when set-dueling is applied to estimate two algorithms.

BIP is likely to activate all warps concurrently. The reason is that

BIP randomly selects the memory blocks to reside in the cache. These cache-resident memory blocks are distributed approximately evenly across the working sets of all warps. The Greedy-Then-Oldest (GTO) algorithm, as the most effective GPU warp scheduling algorithm, targets to minimize the total working set size of an application by prioritizing the currently active warp then the oldest warp. For BIP, the cache-resident memory blocks in the younger warps lead to cache hits followed by misses. BIP thus enables the outstanding requests from more warps and activates more warps concurrently.

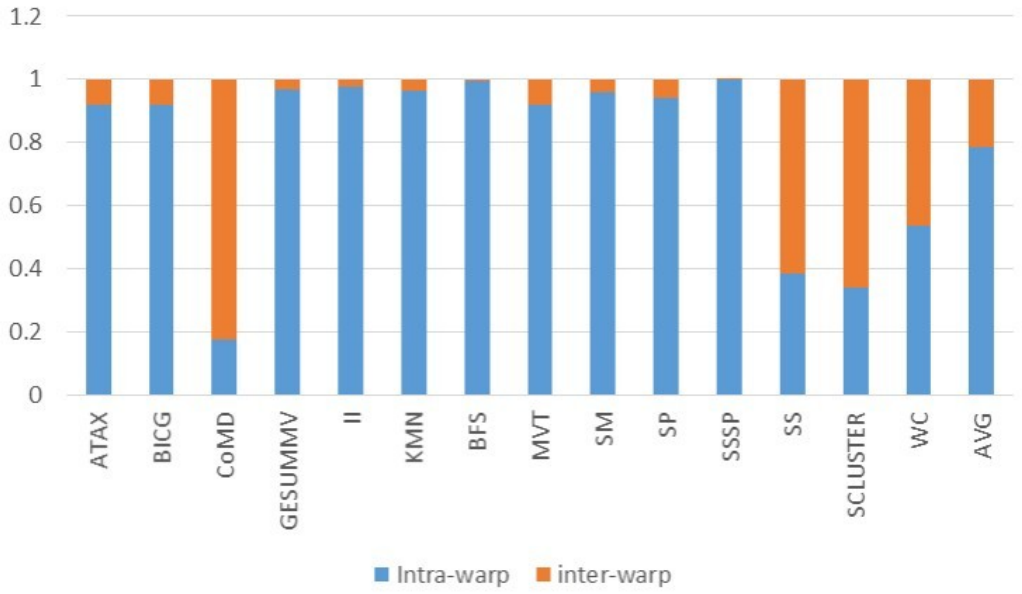


Figure 5.1: Breaking down cache hits into inter-warp and intra-warp reuses

However, most cache block reuses are the intra-warp reuses. Figure 5.1 shows a break down all the cache hits into two categories: inter-warp and intra-warp reuse. A cache hit is recognized as an intra-warp reuse only if

the instruction fetching it into the cache and the instruction hitting it in the cache are from the same warp. Otherwise, it is categorized as an inter-warp reuse. As shown in Figure 5.1, for 10 out of the 14 applications, more than 90% of cache hits are categorized the intra-warp reuse. The other four applications exhibit large proportions of inter-warp reuse. On average, the intra-warp reuses represent 78% of all cache hits. This means that, for most applications, each warp tends to have its own working set. The total working set size of a GPU program is often proportional to the number of active threads. This is the reason that the GTO thread scheduling algorithm aims to minimize the number of active warps. The BIP algorithm tends to activate all warps concurrently and thus leads to a larger aggregated working set. This problem limits the capability of BIP to further improve the cache efficiency and the overall throughput.

Qureshi, et al. [71] propose a set-dueling mechanism which allows multiple replacement/insertion policies to be compared to select one at runtime. DIP relies on the set-dueling mechanism to evaluate the BIP and LRU algorithms on different sets of the same cache and select them adaptively. On the CPU, allowing a subset of cache sets to have a different insertion or replacement policy than the rest of the cache does not change the overall memory reference order. Even when the cache is shared among multiple threads, the CPU cache is often logically partitioned among threads to guarantee the Quality of Service (QoS) of each thread. Consequently, for a logical cache partition dedicated to a thread, the memory reference order does not change.

On the GPU, the L1 cache is not partitioned among threads because the per-thread cache capacity is often less than a cache line. The L1 cache is shared by a large number of hardware thread. A cache miss leads to a fast context switch to another threads. When applying set-dueling mechanism directly to the GPU L1 cache, different sets of the L1 cache on a SM have different insertion or replacement algorithms. Due to the fast thread context switch effect, the memory reference order to the cache can be dramatically different depending on the algorithm applied to the cache. Consequently, it is not feasible to apply the set-dueling mechanism directly to a GPU L1 cache.

As we observe, the root problem of BIP is that its cache block selection and the corresponding insertion algorithm does not match the cache access patterns that are shaped by the thread scheduler. However, if the cache resident data is selected using a scheme adapted to the thread scheduler, performance increases may be achievable. When adapting DIP to a GPU L1 cache, applying different replacement or insertion policies to the different sets of a cache is not feasible. We observe that the cache access patterns of the threads in the same GPU kernel are often very similar. When the threads from the same kernel are loaded to different SMs on a GPU, we can evaluate different algorithms by applying each algorithm in a different SM.

5.2 AgeLRU, a Thread-Scheduling Aware GPU Cache Replacement and Bypassing Policy

5.2.1 Overview

In this section, we propose AgeLRU, a thread scheduling aware cache replacement and bypassing scheme. At replacement, it evicts the least recently used blocks from the youngest warp. This scheme has two goals. (1) it selects the blocks to reside in the cache explicitly considering the effect of the thread scheduling algorithm, so that the total working set size can be minimized. (2) it reserves the cache blocks which tend to lead to more and near cache reuse when selecting victim blocks on cache misses.

As shown in Figure 5.1, most cache block reuses are inter-warp reuses. Each warp tends to have its own working set. Minimizing the number of active warps is likely to minimize the total working set size. Therefore, when selecting memory blocks to reside in the cache, AgeLRU limits all the cache resident blocks to a single warp.

As we discuss in section 3.4.2, The Greedy-Then-Oldest (GTO) algorithm performs best among the common scheduling algorithms such as the Loose Round Robin (LRR) algorithm and the two level (TwoLev) scheduling algorithm. It minimizes the total number of active threads, and the oldest warp tends to have higher priority to execute. The scheduler shapes the memory access pattern directly. For intra-warp reuses in the oldest warps, the reuse distance is less likely to be increased due to context switches to younger warps. Consequently, the AgeLRU algorithm also prioritizes the cache blocks

fetched by the older warp to reside in the cache.

The AgeLRU algorithm can be applied to both the replacement algorithm and the bypassing algorithm. The flowchart of the AgeLRU replacement and bypassing algorithm is described in Figure 5.2. As shown in this figure, the cache control logic selects an invalid cache block or a block fetched by an inactive warp as the replacement victim. If all blocks are fetched by currently active warps, the cache control logic calculates the relative warp age of all cache blocks in the cache set. The replacement logic then selects the least recently used block among the youngest blocks. The bypassing algorithm is an extension of the replacement algorithm. When a block fetched by an active warp is selected as the replacement victim, the replacement logic compares the age of the incoming new request with the age of the victim block, if the new incoming request is younger, the new request is selected to bypass the cache instead of replacing the victim cache block.

Another optimization we apply to the AgeLRU bypassing algorithm is the bypassing traffic optimization. As we demonstrate in Section 4.5, when a memory request bypasses the cache, it does not need to fetch the whole block. This optimization has also been applied to the AgeLRU algorithm. For a detailed discussion of the bypassing traffic optimization see Section 4.5.

Identifying Cache Blocks fetched by Inactive Warps .

The AgeLRU replacement algorithm evicts the least recently used block among the blocks that have youngest relative warp age. The oldest warp tends

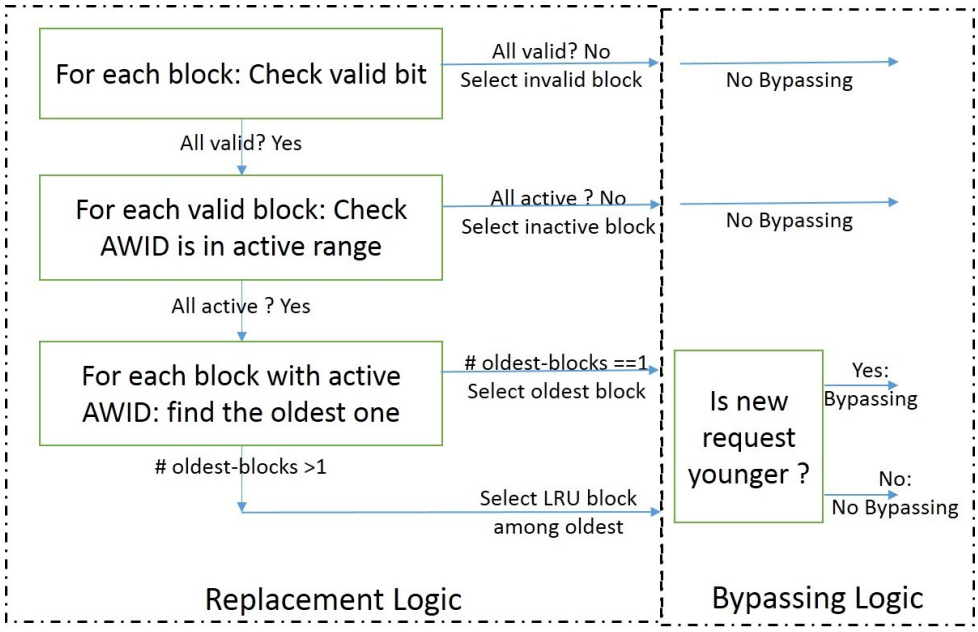


Figure 5.2: Flowchart of AgeLRU Replacement and Bypassing Algorithm

to occupy more cache blocks and achieves a higher cache hit ratio than the younger warps do. However, when the oldest warp completes or reaches a barrier, it is important that the AgeLRU algorithm can detect the blocks that are fetched by currently inactive warps and evict these inactive cache blocks. Otherwise, the cache blocks fetched by the oldest warp are never going to be utilized by other warps. We propose to keep tracking the oldest active warp id (Oldest-AWID) and the youngest active warp id (Youngest-AWID) at each SM, and allow each cache block to store its fetching warp's active warp ID (AWID). Comparing the AWID value of a cache line with the Oldest-AWID and the Youngest-AWID can identify the cache blocks fetched by currently inactive warps.

This mechanism can not identify the inactive warp when the non-oldest warp completes before the oldest one does. An optimal solution is to maintain a list of active WID on each SM and thus any ID not in the list can be identified as inactive. However, it requires extra storage and extra energy to look up the table. We believe that the mechanism keeping the oldest and youngest warp id is able to achieve a similar effect as the full list of warp ID does. The reason is two-fold, (1) in a regular GPU application, most warp finishes in the order as they are fetched, because the scheduler prioritizes the older warps. (2) we maintain the oldest warp ID as the oldest active warp ID. The cache blocks fetched by the oldest warp has the highest priority to reside in the cache at replacement. When there is a non-oldest warp finishes, the blocks it fetched still have a lower priority and can be evicted by the requests from the older warp.

5.2.2 Implementing AgeLRU

Figure 5.3 illustrates the implementation of the AgeLRU algorithm in a streaming multiprocessor (SM). At a high level, each SM maintains the Oldest active warp ID (Oldest-AWID) register (❶ in Figure 5.3) and the youngest active warp ID (Youngest-AWID) register (❷ in Figure 5.3).

The Active Warp ID (AWID) bits are added to the warp status field of each warp (❸ in Figure 5.3), and to the memory request packet format, as well as per-line storage of AWID meta-data for the L1 cache (❹ in Figure 5.3). Both Oldest-AWID and Youngest-AWID are implemented as 8 bit wrap-

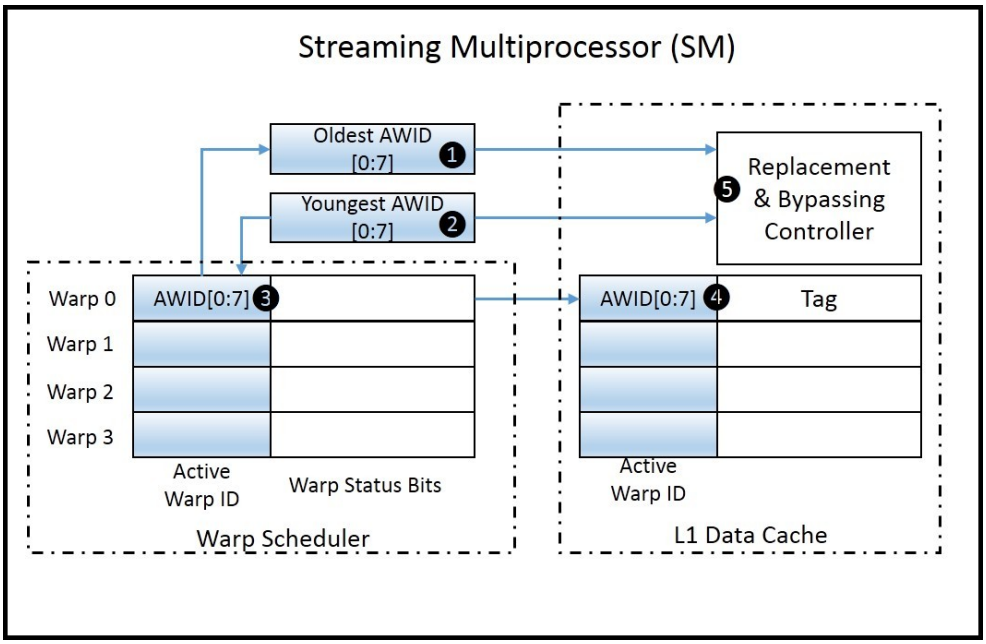


Figure 5.3: AgeLRU Implementation

around counters. Oldest-AWID gets the AWID from the next active warp when the oldest warp completes or reaches a barrier. Youngest-AWID increases by one every time a new warp is fetched. Both the Oldest-AWID register and the Youngest-AWID register are implemented as wrap-around counters. When either of the two registers reaches the maximum value of the 8-bit registers i.e. 255), it starts from zero again.

Identifying an inactive WID requires to judge whether a given WID falls in the range between $[Youngest - AWID, Oldest - AWID]$. As shown in Figure 5.4, when the youngest AWID value does not reach the maximum value (255), a given WID is active when:
$Youngest - AWID <= WID <= Oldest - AWID$.

When the youngest AWID value wraps around, a given WID is active when:

$Youngest - AWID >= WIDorWID >= Oldest - AWID.$

Youngest-AWID and Oldest-AWID can also determine the relative age of a given active WID. The relative warp age of a cache line with an active AWID, can be calculated by

$relative - warp - age = (AWID - Oldest - AWID)\%255$

AWID may be transmitted with any memory request or message to propagate status. The cache replacement and bypassing control logic (❺ in Figure 5.3) uses AWID bits from each cache line, Oldest-AWID and Youngest-AWID to select the replacement victim block.

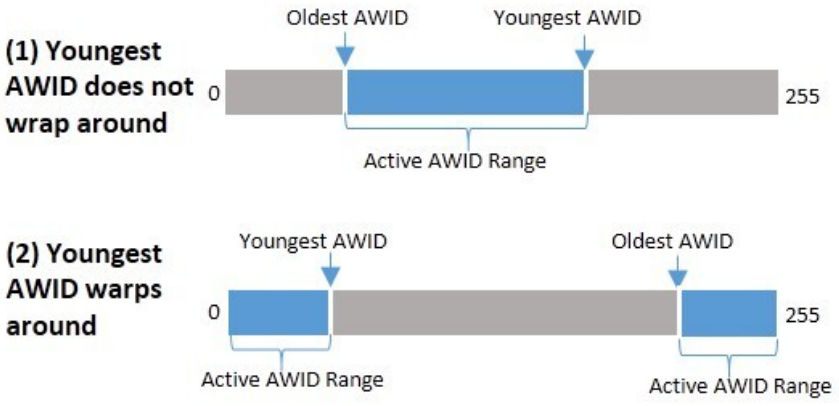


Figure 5.4: Identify Inactive WID with the wrap-around counters

5.3 Dynamic AgeLRU

For an application experiencing a cache thrashing problem, the AgeLRU algorithm is able to keep a fraction of the working set in the cache and thus improve the overall performance. However, for non-cache-thrashing applications, the AgeLRU algorithm sometime hurts cache efficiency and the overall throughput. CPU cache trashing-resistant algorithms have a similar problem, a common solution is to dynamically switch between the thrashing-resistant algorithm and the default algorithm such as LRU. For instance, BIP [71] randomly selects memory blocks to reside in the cache to alleviate the cache thrashing problem. DIP [71] relies on the set-dueling mechanism to select BIP or LRU dynamically. The set-dueling mechanism [71] has been applied to other CPU cache algorithms to dynamically choose between two cache algorithms [31] [97]. As we discuss in 5.1, the set-dueling mechanism is not feasible to be applied directly to the GPU cache system to dynamically select the cache algorithm that best fits an application.

Parallel Voting Mechanism

We propose to apply a parallel voting mechanism to select the AgeLRU algorithm and LRU algorithm adaptively. Figure 5.5 shows the state machine of the parallel voting mechanism. It also demonstrates the state machine with an example showing the L1 cache algorithms of all SMs at each state of the parallel voting state machine. The state machine of the parallel voting mechanism

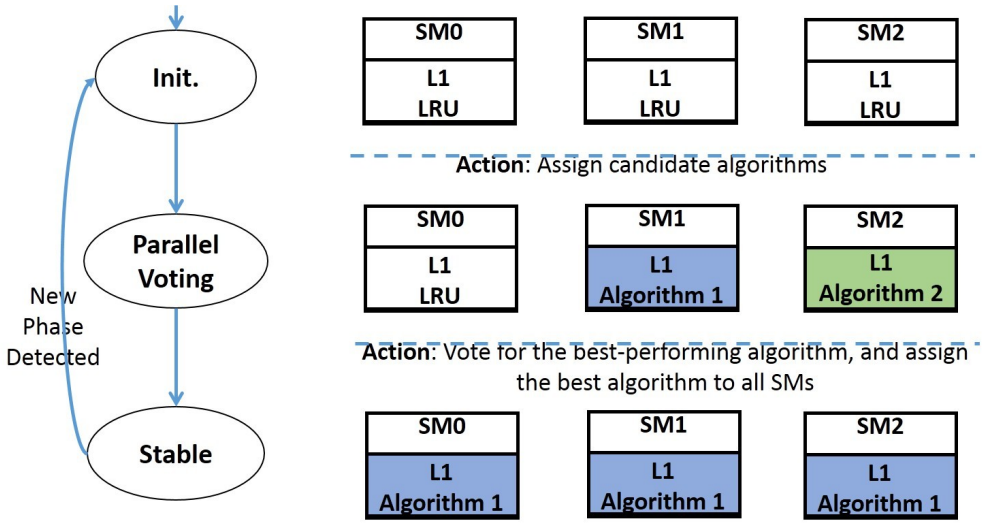


Figure 5.5: Parallel Voting Mechanism: the state machine and an example showing the L1 cache algorithms of all SMs at each state of the parallel voting state-machine

is described as follows. As shown in Figure 5.5, when a new program phase is detected or the program just starts, the parallel voting state machine applies the candidate algorithms each on its own SM. It allows the other SMs to use the default LRU replacement algorithm. At next sampling, the throughput of the SM that applies the LRU algorithm is compared to the throughput of the SMs that apply the other candidate algorithms. The algorithm leading to the highest per-SM throughput is voted to be the best algorithm. It is applied to all SMs until another new program phase comes.

To implement the Dynamic-AgeLRU algorithm, the parallel voting mechanism takes the AgeLRU algorithm as the candidate algorithm. It selects the AgeLRU algorithm and the default LRU algorithm adaptively.

5.4 Results

In this section we present the results for the AgeLRU/Dynamic-AgeLRU algorithms. The AgeLRU and Dynamic-AgeLRU algorithm can be applied as both the replacement and the bypassing algorithms. Additionally, the bypassing traffic optimization can be applied to the bypassing algorithm. Therefore, for each of the AgeLRU and the Dynamic-AgeLRU algorithm, we evaluate three configurations: (1) replacement-only, (2) bypassing, and (3) bypassing with traffic optimization.

Baseline

In our experiment, the LRU algorithm is applied to the baseline architecture which is configured as in Table 3.4. We also compare the AgeLRU and Dynamic-AgeLRU algorithm with the classic CPU cache thrashing-resistant algorithms, adapting the BIP mechanism to the GPU L1 cache replacement. 1/32 of memory blocks are assigned with higher priority to reside in the cache (as in the MRU state), while the other memory blocks are assigned with lower priority (as in the LRU state). The original DIP algorithm relies on the set-dueling mechanism to select BIP and LRU adaptively. Since set-dueling does not work on GPUs, we approximate the DIP algorithm with the optimal-DIP algorithm where the replacement algorithm is selected off-line to maximize the performance. Similar to the AgeLRU and Dynamic-AgeLRU algorithm, the BIP and DIP algorithm can also be configured as Replacement-only, Bypassing, and Bypassing with traffic optimization. We compare the AgeLRU and

the Dynamic-AgeLRU algorithm to the BIP and DIP algorithm on each of the configurations.

Overall

We compare the AgeLRU algorithm with the baseline and BIP algorithm across a collection of key cache-sensitive benchmarks. Next, in order to evaluate the AgeLRU algorithm with the cache insensitive benchmarks, we evaluate AgeLRU with a large collection of benchmarks. These benchmarks exhibit variable sensitivities to the cache algorithm. We observe that AgeLRU degrades the performance of 3 benchmarks. Next, we evaluate the Dynamic-AgeLRU algorithm with both the cache-sensitive benchmarks and the three AgeLRU unfriendly benchmarks. Dynamic-AgeLRU is compared to the baseline and the DIP algorithm. At the end, we summarize our results and conclusions.

5.4.1 Evaluating AgeLRU

In this section, we compare the results of three sets of results between the AgeLRU algorithm and the BIP algorithm used as: (1) replacement algorithms, (2) bypassing algorithms, and (3) bypassing algorithms with traffic optimization applied.

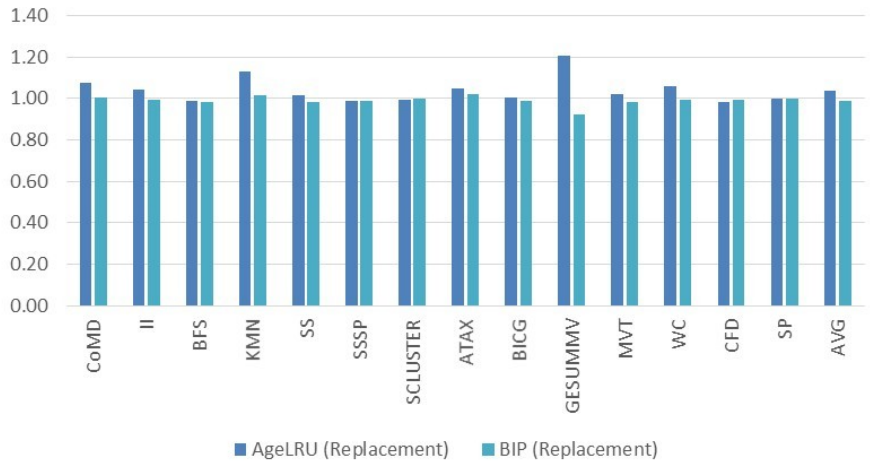


Figure 5.6: Speedup of AgeLRU replacement and BIP replacement algorithms

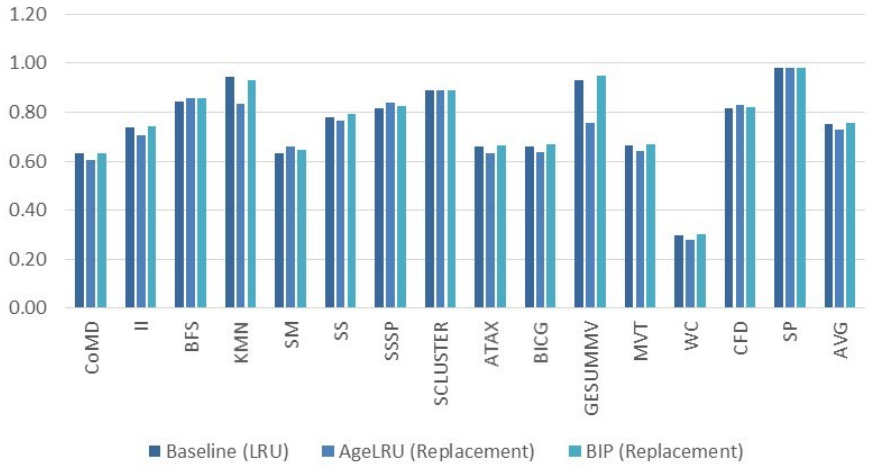


Figure 5.7: L1 miss rate of LRU, AgeLRU replacement and BIP replacement algorithms

5.4.1.1 Replacement algorithm

Figure 5.6 compares the speedup of the AgeLRU replacement and the BIP replacement algorithm. All the results have been normalized to the baseline which applies the LRU algorithm as its L1 cache replacement algorithm. As shown in Figure 5.6, among the 14 benchmarks, the AgeLRU based replacement algorithm is able to achieve noticeable speedups (a speedup not smaller than 5%) for 6 benchmarks. It improves the throughput of CoMD, II, KMN, ATAX, GESUMMV and WC by 7%, 5%, 13%, 5%, 21% and 6% respectively. On average, AgeLRU achieves a speedup of 4% across the 14 cache sensitive benchmarks. It demonstrates that for these benchmarks experiencing inter-thread cache contention problems, the AgeLRU algorithm is able to alleviate this contention and achieve a higher throughput. For the other benchmarks, the AgeLRU replacement algorithm can not help the throughput. One of the major reasons is that these benchmarks not only experience the cache thrashing problem, but also the chip resource saturation problem, particularly, either the NoC bandwidth saturation problem or the cache line reservation failure problem.

Figure 5.7 compares the L1 miss rate of the LRU, AgeLRU replacement and BIP replacement algorithms. As shown in this figure, the L1 miss rate reductions that the AgeLRU algorithm achieves correlate with the speedup it achieves very well. For the 6 benchmarks where the AgeLRU algorithm enables higher throughput, the L1 miss reduction is 2%, 3%, 11%, 3%, 17% and 2% respectively. The maximum speedup that the AgeLRU algorithm

achieves is 21% for GESUMMV. GESUMMV is also the benchmark for which the AgeLRU algorithm achieves the largest L1 miss reduction of 17%.

In contrast to the AgeLRU algorithm, the BIP based replacement algorithm does not achieve noticeable speedup for most of the benchmarks. The L1 miss reduction it enables is also not noticeable. The reason is that the BIP algorithm selects the resident cache blocks randomly, which activates most of the warps and enlarges the total working set.

We demonstrate this effect with a case study of KMN. For KMN, the LRU algorithm leads to a L1 miss rate of 94%, the BIP algorithm reduces the miss rate to 93%. By contrast, the AgeLRU algorithm enables a reduction of 11% and leads to a miss rate of 83%. Figure 5.8 compares the L1 hit number of KMN for three algorithms: LRU, AgeLRU and BIP. All numbers are normalized to the total number of L1 hits that LRU enables. The total number of cache hits that the AgeLRU enables is 2.9X that of LRU. The BIP enables 1.24X cache hits compared to the LRU. More importantly, this figure breaks down L1 cache hits according to the relative age of the warp that issues the request. As shown in this figure, BIP enables all the warps to get a little bit more L1 cache hits. It is because that BIP selects the cache resident blocks randomly, they are distributed among all warps. In contrast to the BIP algorithm, the AgeLRU algorithm prioritizes the working set of the older warps to be resident in the cache. The oldest warp achieves many more cache hits than the younger warps. The oldest four warps account for 49% of all the cache hits. Compared to the BIP algorithm, the AgeLRU algorithm is

less likely to activate the younger warps.

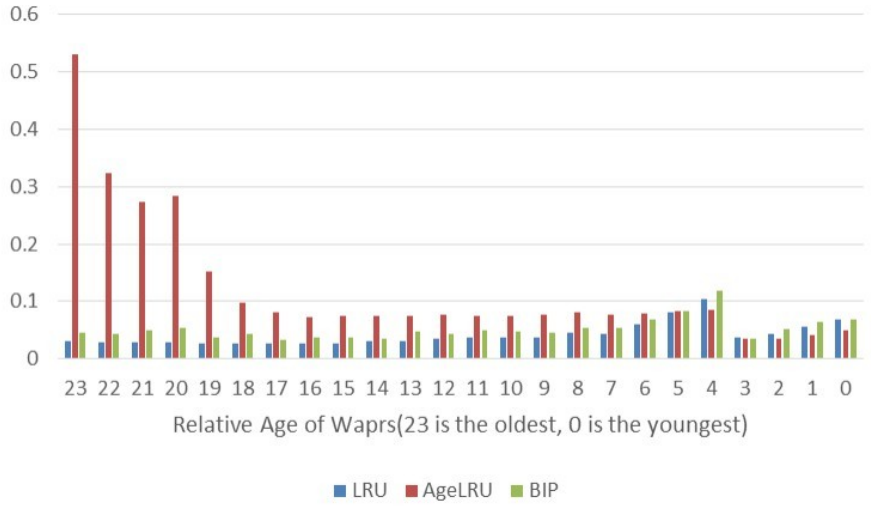


Figure 5.8: A case study of KMN, comparing LRU, AgeLRU and BIP. Breaking down the L1 cache hit number according to the relative age of the warp that issues the request. (**Note 1:** The relative age here is the rank when all the active warps on a scheduler are sorted by the fetching time. The youngest warp's relative age is 0 and the oldest one is 23. **Note 2:** All numbers are normalized to the total number of L1 hit that LRU enables.)

5.4.1.2 Bypassing algorithm

Figure 5.9 compares the speedup of the AgeLRU bypassing and the BIP bypassing algorithm. All the results have been normalized to the baseline which applies the LRU algorithm as its L1 cache replacement algorithm. Compared to the replacement only algorithm, the bypassing algorithm enables the bypassing traffic not to reserve any cache blocks before going to next level of the memory hierarchy. The reservation failure of cache blocks can be avoided. As shown in this figure, SCLUSTER achieves a speedup of 55%, while the

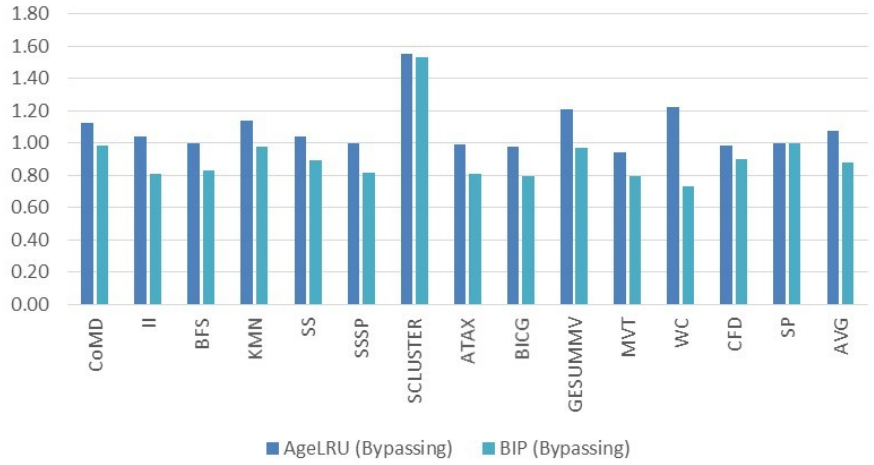


Figure 5.9: Speedup of AgeLRU bypassing and BIP bypassing algorithms

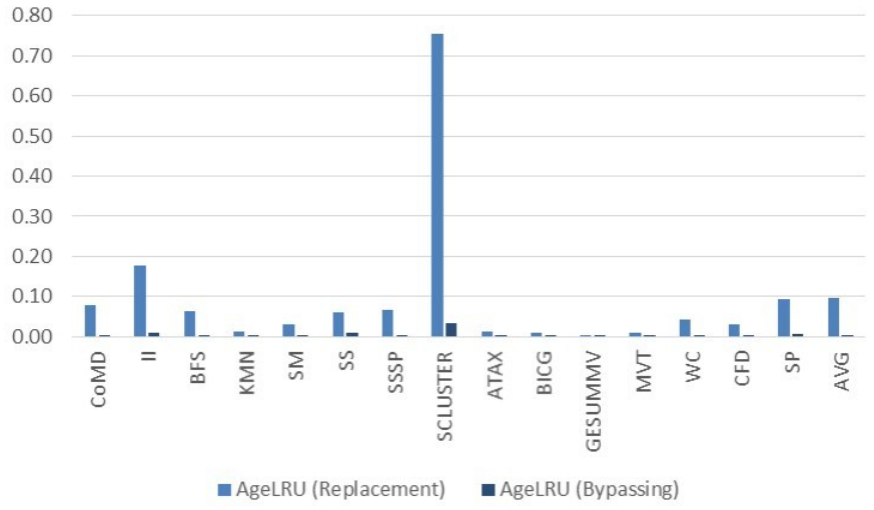


Figure 5.10: Memory pipeline stall ratio due to L1 block reservation failure: AgeLRU replacement and AgeLRU bypassing

AgeLRU replacement algorithm does not improve its performance. Figure 5.10 shows the memory pipeline stall ratio due to the L1 block reservation failure for the AgeLRU replacement and AgeLRU bypassing algorithm. When the AgeLRU based replacement only algorithm is applied, for SCLUSTER, most of its memory pipeline stalls are caused by L1 block reservation failures. The AgeLRU bypassing algorithm enables bypassing traffic which does not need to reserve the L1 blocks. The L1 reservation failure caused memory pipeline stalls drop significantly. Consequently, the AgeLRU bypassing algorithm enables SCLUSTER to achieve a speedup of 55% .

5.4.1.3 Bypassing algorithm with bypassing traffic optimization

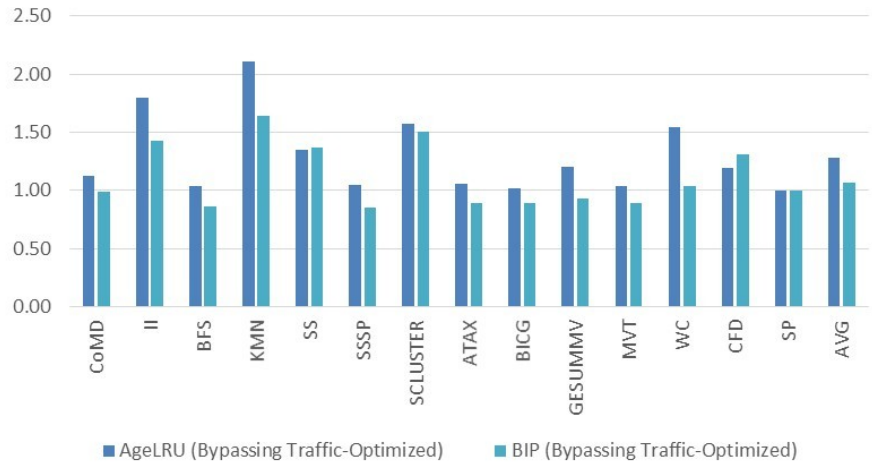


Figure 5.11: Speedup of AgeLRU bypassing and BIP bypassing (Both with bypassing traffic optimization)

The bypassing traffic optimization is also applied to the AgeLRU bypassing algorithm. When a memory request bypasses the cache, it only fetches

the portion of the cache block it needs instead of the whole block. Figure 5.11 compares the speedup of the AgeLRU bypassing and the BIP bypassing algorithm when the bypassing traffic optimization is applied. Similar to the BIP bypassing algorithm, the BIP bypassing algorithm with the traffic optimization degrades the performance for several benchmarks including BFS, SSSP, ATAX, BICG and GESUMMV. However, when the traffic optimization is applied with the BIP algorithm, almost all the requests are optimized. As a result, the BIP algorithm also achieves significant speedups for II, KMN, SS and CFD. However, the AgeLRU bypassing algorithm with the traffic optimization achieves even higher speedups than the BIP based algorithm. When the bypassing traffic optimization is applied, the AgeLRU bypassing algorithm enables 10 of the 14 benchmarks to achieve significant speedups. It deliveries an average speedup of 28% across the 14 cache-sensitive benchmarks.

Compared to the bypassing algorithm without the traffic optimization applied, the traffic optimization enables the applications to utilize the NoC bandwidth more effectively and reduce the NoC network latency. Furthermore, it reduces the NoC congestion rate and thus other requests are less likely to stall due to the NoC congestion. To investigate the NoC congestion, we investigate the memory request round trip latency and the speedup of the AgeLRU bypassing algorithm with/without bypassing traffic optimization.

Figure 5.12 compares the speedup of the AgeLRU bypassing algorithm and with and without bypassing traffic optimization. The optimization enables higher speedup for 6 benchmarks including II, KMN, SM, SS, WC and CFD.

Figure 5.13 shows the average round trip latency of the memory requests of the AgeLRU bypassing algorithm with and without bypassing traffic optimization. As shown in this figure, the optimization enables II, KMN, SM, SS, WC and CFD to achieve a significant reduction in the average round trip latency. These two figures demonstrate that there is a strong correlation between the round trip latency reduction and the throughput improvement.

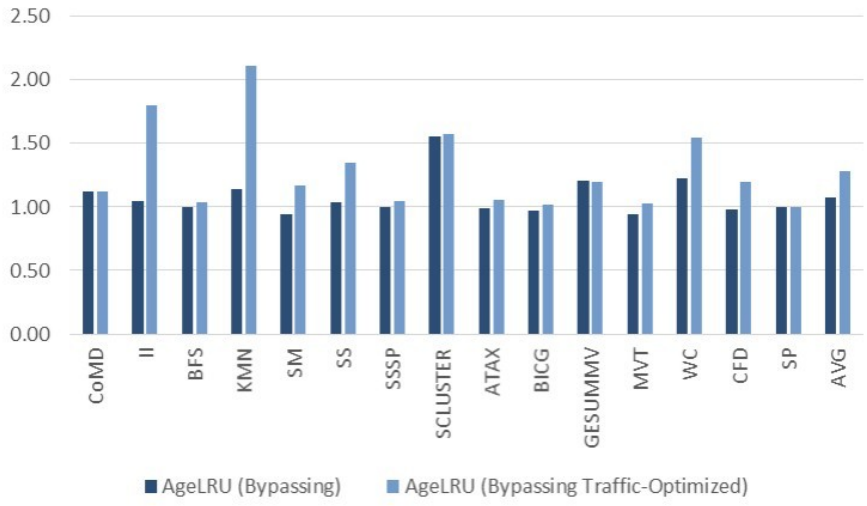


Figure 5.12: Comparing speedup of AgeLRU bypassing with/without traffic optimization

5.4.1.4 Evaluating AgeLRU with varieties of benchmarks

After demonstrating that the AgeLRU algorithm can enable significant speedup for the cache-sensitive applications in Section 5.4.1.1, 5.4.1.2 and 5.4.1.3, we evaluate the approach with a large collection of benchmarks with varieties

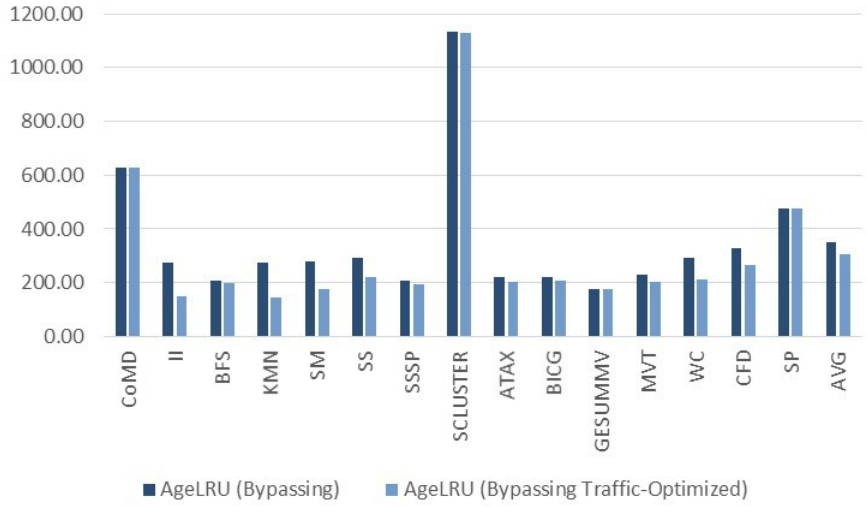


Figure 5.13: Comparing memory requests round trip latency with/without traffic optimization

of sensitivity to the cache algorithms. Figure 5.14 shows the speedup of the AgeLRU replacement and bypassing with/without traffic optimization for a large collection of benchmarks. As shown in this figure, there is a subset of benchmarks that achieves significant speedup with the AgeLRU algorithms. Most of them are the cache-sensitive benchmarks we evaluate in previous sections. There are many applications that do not respond to the cache replacement and bypassing algorithms. It is also noticeable that there are 3 benchmarks for which the AgeLRU algorithms decreases the throughput. These 3 benchmarks, GEMM, 3MM and ATAXI favor the LRU algorithm instead of the AgeLRU algorithm. As we discuss in Section 5.3, we develop the Dynamic-AgeLRU to select the LRU and AgeLRU algorithm adaptively. The Dynamic-AgeLRU algorithms are evaluated in next section.

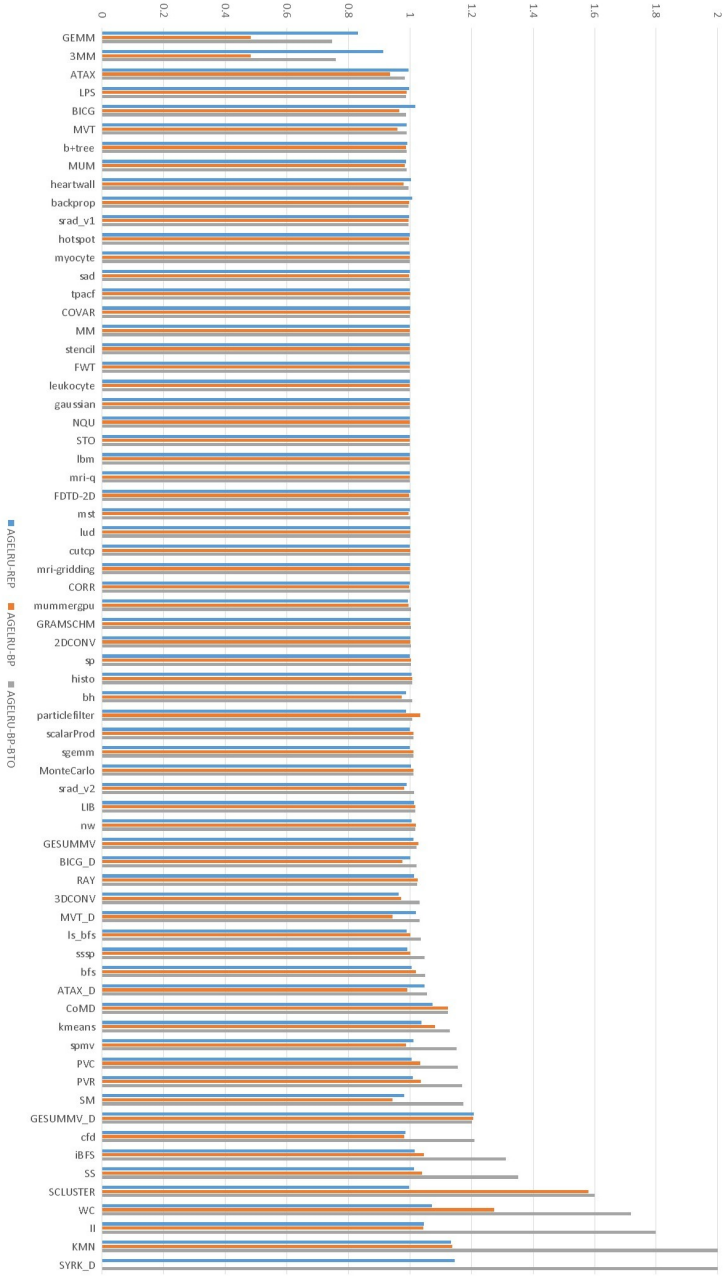


Figure 5.14: Speedup of AgeLRU Replacement and Bypassing with/without traffic optimization

5.4.2 Evaluating Dynamic-AgeLRU

In this section, we evaluate the Dynamic-AgeLRU replacement, bypassing and bypassing with traffic optimization algorithms for both the cache-sensitive benchmarks and the three benchmarks that get performance degradation with the AgeLRU algorithms.

The results are not only compared to the baseline LRU algorithm but also to the CPU cache thrashing-resistant algorithm DIP. We approximate the DIP algorithm with optimal-DIP where the LRU or BIP algorithm is selected off-line to maximize the best performance.

Figure 5.15, Figure 5.16 and Figure 5.17 summarize the speedup of the 4 schemes: AgeLRU, Dynamic-AgeLRU, BIP and optimal-DIP. These figures show the 3 configurations of each of the schemes: replacement only, bypassing and bypassing with traffic optimization. 3MM, ATAXI, GEMM are the three benchmarks that get performance degradation with AgeLRU algorithms. As shown in these figures, the Dynamic-AgeLRU algorithms are able to avoid the performance degradation by choosing the LRU algorithm instead of the Age-LRU algorithm for these three benchmarks. The largest performance drop for Dynamic-AgeLRU replacement, bypassing and bypassing with traffic optimizations are 3%, 6%, 3% respectively.

For most of the other benchmarks, the Dynamic-AgeLRU algorithm that automatically selects either the AgeLRU or LRU algorithm is competitive with the BIP algorithms. GESUMMV is the exception where Dynamic-

AgeLRU always selects LRU instead of AgeLRU, although the AgeLRU replacement algorithm enables this benchmark to achieve a speedup of 21%. The reason is that the Dynamic-AgeLRU algorithm always evaluates the AgeLRU algorithm on SM0. GESUMMV does not have sufficient CTAs to keep all SMs busy and SM0 is always idle. As a result, the Dynamic-AgeLRU algorithm does not have an opportunity to evaluate the AgeLRU algorithm so it always select the LRU algorithm.

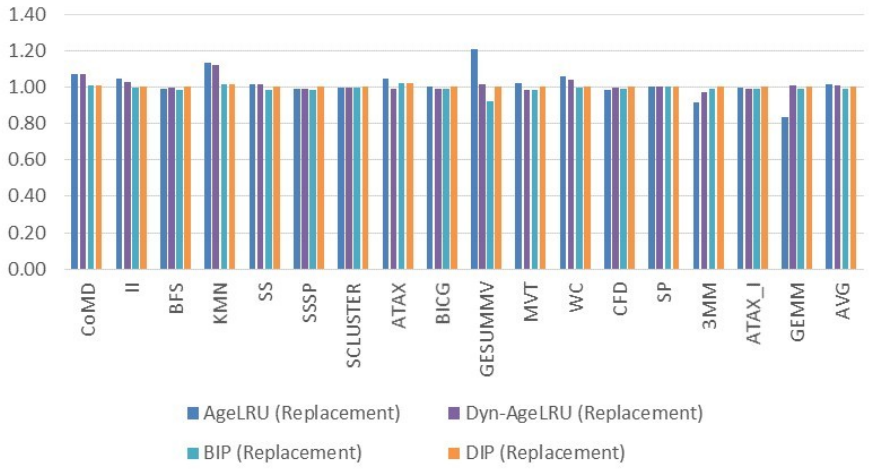


Figure 5.15: Speedup of AgeLRU, Dynamic-AgeLRU, BIP and Optimal-DIP replacement algorithms

5.5 Summary

In this chapter, we propose the AgeLRU and Dynamic-AgeLRU based replacement and bypassing algorithms that adapt to the GPU thread scheduling algorithm. The bypassing traffic optimization has also been applied to utilize the NoC more effectively.

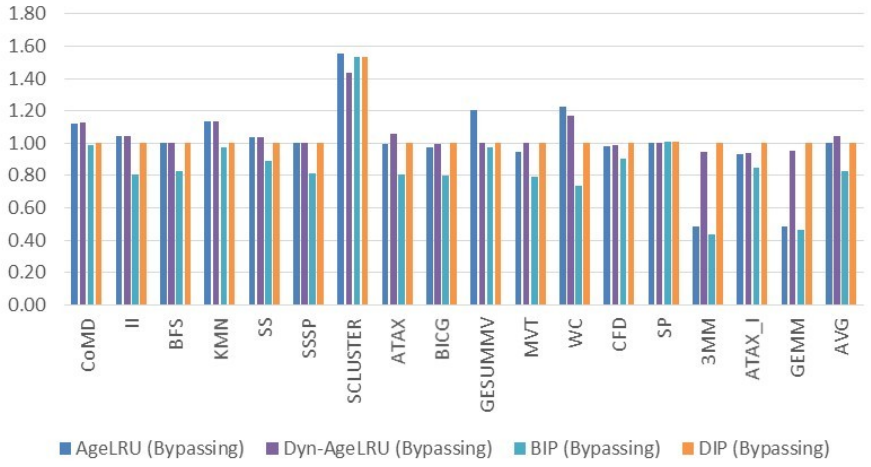


Figure 5.16: Speedup of AgeLRU, Dynamic-AgeLRU, BIP and Optimal-DIP bypassing algorithms

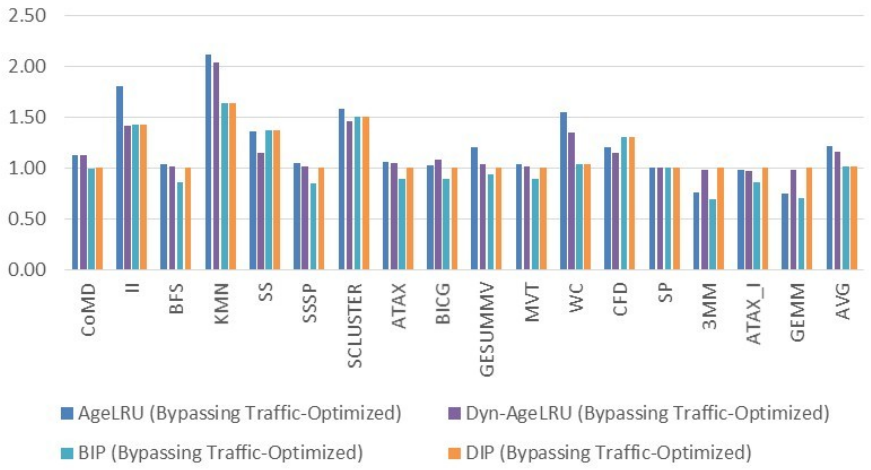


Figure 5.17: Speedup of AgeLRU, Dynamic-AgeLRU, BIP and Optimal-DIP bypassing algorithms (all with bypassing traffic optimization)

The AgeLRU replacement algorithm is able to reduce the inter-thread cache contention and thus achieve throughput improvement. The AgeLRU based bypassing algorithm provides an extra benefit beyond the AgeLRU replacement only algorithm. It reduces the number of requests that reserve cache lines. This turns out be helpful for the applications that experience cache line reservation failure, such as SCLUSTER. The AgeLRU based bypassing algorithm with the bypassing traffic optimization shows significant speedup not only over the baseline LRU algorithm but also over CPU cache thrashing-resistant algorithms such as DIP.

To avoid the AgeLRU algorithm degrading the throughput for the non-thrashing applications, we propose Dynamic-AgeLRU to select the AgeLRU algorithm and the LRU algorithm adaptively at the beginning of each program phase.

Our results show that Dynamic-AgeLRU can avoid degrading the performance of non-thrashing applications by selecting the LRU algorithm. We also demonstrate that the Dynamic-AgeLRU algorithms are able to achieve the majority of the performance improvements that the AgeLRU algorithms can achieve for most applications.

In summary, the AgeLRU and Dynamic-AgeLRU algorithms are effective thrashing-resistant GPU L1 cache algorithms. The Dynamic-AgeLRU algorithm not only enables the thrashing applications to achieve significant speedups but also does not hurt the non-thrashing applications. Its advantage not only includes its capability to improve the performance with a low hard-

ware cost, but also includes the fact it requires no programmer intervention.

Chapter 6

Reuse-Prediction-based Scheduler-Aware GPU Cache Replacement Algorithm

In this chapter, we propose a Reuse-Prediction-based cache Replacement scheme (RPR) for a GPU L1 data cache to address the intra-thread cache pollution problem. Unlike the AgeLRU algorithm that prioritizes cache blocks by the age of a warp, this scheme identifies and prioritizes the near-reuse blocks and high-reuse blocks to maximize the cache efficiency so that it can achieve further throughput improvement beyond the AgeLRU mechanism.

The RPR mechanism can approximate the reuse-distance-based algorithm, the counter-based algorithm, and the AgeLRU algorithm with various cache block score functions. Compared to the corresponding CPU cache pollution-resistant algorithms, the difference of RPR is two-fold. (1) RPR leverages the GPU thread scheduling priority (i.e. the relative age of a warp) together with the fetching PC to generate the signature, which is the index of the prediction table. (2) RPR calculates the average reuse distance of each group to tolerate unpredictable thread interleaving.

Compared to the AgeLRU algorithm, the experimental results show that the RPR algorithm enables a throughput improvement of 5% on average

for the regular applications, and a speedup of 3.2% across the benchmarks we evaluate.

For irregular problems, the AgeLRU algorithm and the default LRU algorithm outperform the reuse-prediction-based replacement algorithm. The reason is that the AgeLRU algorithm prioritizes memory blocks at a coarse granularity and can tolerate more irregularities. A dynamic RPR mechanism can select the RPR algorithm, LRU algorithm or AgeLRU algorithm adaptively to avoid hurting irregular applications. Consequently, the dynamic-RPR mechanism can select the LRU or the AgeLRU algorithm for the irregular programs while the RPR algorithm can be selected for regular applications. The dynamic RPR mechanism can be implemented with the parallel voting mechanism similar to the Dynamic-AgeLRU algorithm we discuss in Section 5.3. We leave dynamic-RPR as future work.

The rest of the chapter is organized as follows. First, we discuss the feasibility of adapting CPU pollution-resistant algorithms to the GPU L1 cache. Next we introduce the implementation details of the reuse-prediction-based cache replacement algorithm (RPR). At the end, we analyze the experimental results.

For many memory intensive GPU applications, the cache is thrashed by large number of active hardware threads. In this case, cache thrashing becomes the major problem. In this chapter we apply the optimal static CTA throttling as the first step, which creates the performance baseline of our research. All the new policies we propose are applied beyond the CTA throttling.

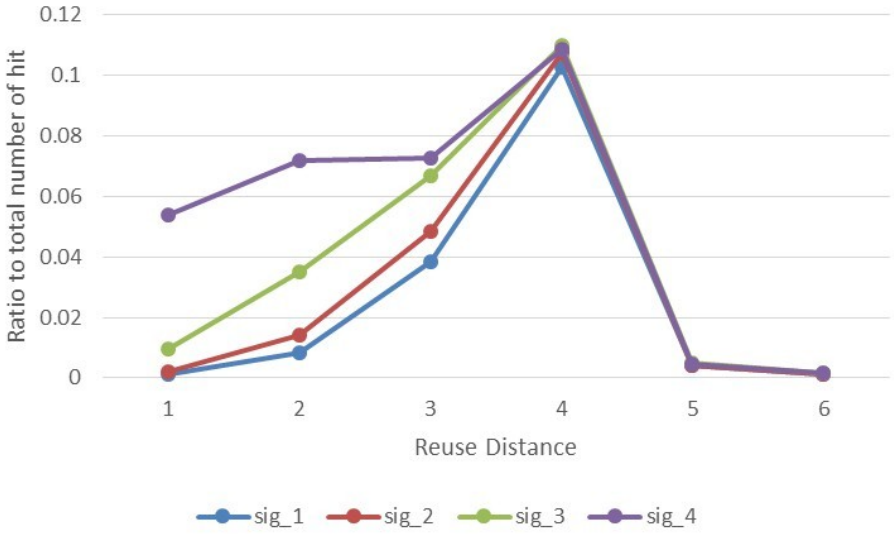
After CTA-throttling is applied appropriately, the cache pollution problem becomes the dominant factor that still degrades the cache efficiency. As our key contribution, we propose GPU specific cache replacement algorithms based on reuse prediction to alleviate the cache pollution problem.

6.1 Motivation

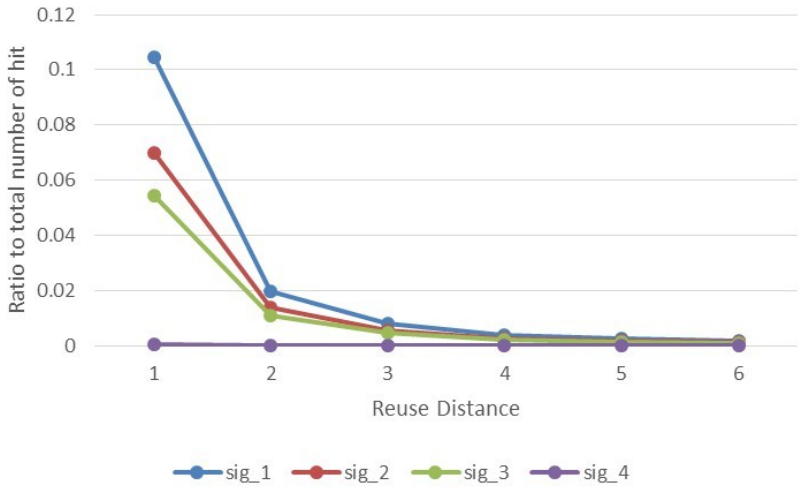
As with the CPU cache, the GPU cache also experiences cache thrashing, cache pollution and dead block problems. Researchers have proposed many CPU cache pollution-resistant algorithms.

One big category of these algorithms is the signature-based reuse behavior prediction enhanced replacement algorithm. Examples include the counter-based replacement algorithm [44], the Re-reference Interval Prediction (RRIP) [31], the Signature-based Hit Predictor (SHiP) [97], the reuse-distance prediction based cache replacement policy [42] and cache burst based dead block prediction [52]. The reuse prediction schemes categorize all memory quests into groups by summarizing their attributes into signatures. The signature in general is hashed from simple attributes of the request, such as the last touching PC, the fetching PC, or the memory region address etc. The requests in a group that shares a signature are expected to share similar reuse behaviors. The predictor anticipates that the reuse behavior of a new request will be the same as the previous requests that share a signature with the new one.

It is desirable to apply these algorithms to the GPU cache to address



(a) KMN



(b) SCLUSTER

Figure 6.1: Reuse distance distribution of memory blocks in same group sharing a signature. Case study of KMN and SCLUSTER. The 4 signatures that represent the largest 4 groups are shown in the figure.

the pollution problem. However, we observe that there are two problems that limit the feasibility of applying these algorithms directly.

1. **Without considering the effect of the thread scheduling, a high accuracy reuse prediction is not achievable.**

   The GPU thread scheduler has a significant effect on the GPU cache access pattern. A reuse prediction must consider the effect of the thread scheduling. For instance, reuse predictors often generate a signature for each cache block based on the program counter (PC) of the instruction that fetches or last touches the block. Cache blocks that share the same signature are expected to show similar reuse behavior, which has been proven to be effective for a CPU cache. However, in the case of GPUs, the PC of a memory instruction can not fully decide the reuse behavior of the related cache block. The thread scheduler has major effect on the thread interleaving order. Prior work shows that the Greediest Than Oldest (GTO) is so far the best thread scheduler in terms of performance. By prioritizing the older warps, the scheduler only issues instructions from the younger warp when the older warps do not have any ready instructions. Consequently, the memory accesses issued from older warps are more likely to be reused in the near future. The instance of the instruction in the older warps normally has higher scheduling priority, and is more likely to be reused sooner.

2. **The reuse distance and count of the memory requests sharing**

a signature can diverge in a range, instead of mostly concentrating on one value as on CPUs.

The reason for this is that the GPU cache access stream is highly affected by the thread interleaving which is not stable. A CUDA program is very finely threaded. The GPU cache access stream is a mix of the requests from many threads. GPUs allow fast context switch among these threads on long latency operations such as a cache misses and complex arithmetic instructions. The interleaving among threads is not always recurring and predictable. The interleaving order of these threads is affected by many GPU specific affects. The hardware thread scheduler has a direct effect on the thread interleaving, but unexpected latency could also change the interleaving order. The unexpected latency could include instruction stalls due to hardware hazards and shared memory bank conflicts.

As a result, if two memory requests are sharing similar reuse patterns according to the source code, their actual reuse distance or reuse count could diverge. Figure 6.1 shows the reuse distance distribution of memory blocks in a group that shares a signature. For KMN and SCLUSTER, the 4 signatures representing the largest 4 groups are shown in the figure. Almost all the groups show a certain amount of divergence. For example, the signature-one in KMN, although the largest portion of the requests in the group exhibit a reuse distance of four, the average reuse distance of this group is three.

In order to optimize the GPU cache replacement algorithm, the thread scheduling algorithm must be taken into account. In other words, the replacement algorithm needs to consider the access patterns generated by specific threading algorithms.

6.2 Reuse-prediction-based Replacement Algorithms

In this section, we introduce the reuse-prediction-based cache replacement algorithm (RPR). Compared to the CPU pollution-resistant algorithms, the RPR algorithm leverages not only the well studied GPU cache replacement techniques, but also takes advantage of GPU-specific attributes. The RPR algorithm relies on two new features to overcome the problems we discuss in Section 6.1. First, the signature is hashed from not only the PC as in CPU cache algorithms, but also the relative age of the fetching warp. Second, the RPR algorithm keeps track of the average reuse distance and reuse count of each request group so that it can tolerate a certain amount of divergence.

RPR predicts the reuse distance and the reuse count of every cache block in the set when selecting a victim for replacement. Then it evicts the one that has been predicted to be reused farthest in future.

The RPR approach consists of three major parts as shown in Figure 6.2: (1) the reuse distance and count sampler, (2) the reuse predictor, and (3) the reuse-prediction enhanced cache controller. At a high level, the sampler is a small tag array with counters. It tracks the memory access to a portion of the cache sets. The sampler is responsible for collecting the reuse distance and

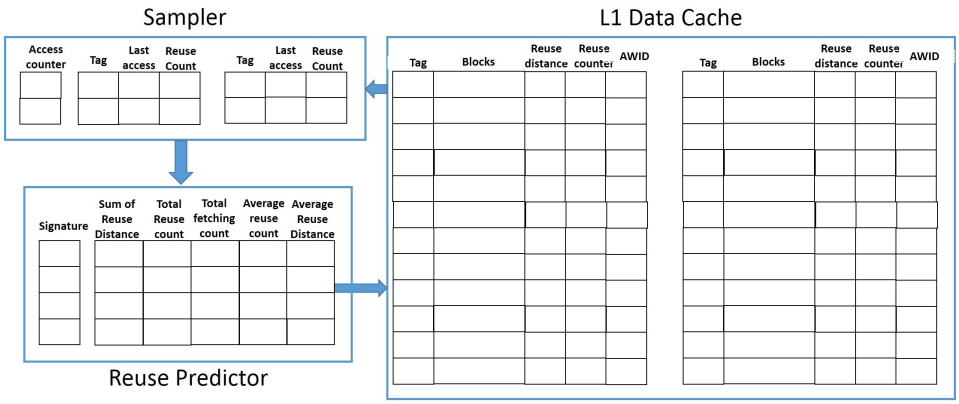


Figure 6.2: Overview of implementing the reuse-prediction-based Replacement mechanism

reuse count of the sampled cache blocks and sending the reuse information to the reuse predictor to update the prediction history table. The reuse predictor maintains the history of the reuse distance and count of every emerged signature and predicts the reuse behavior of any new memory block. The reuse prediction enhanced replacement controller initializes each new fetched block with the predicted reuse distance and count, maintains the reuse information at every cache access, and selects the victim block based on the reuse distance, the reuse count and the LRU combined. The implementation details of the three major components of the new approach are described in the rest of this section.

6.2.1 Reuse Distance and Reuse Count Predictor

Reuse distance is defined as the number of accesses to the cache set between two consecutive accesses to the same cache block. Reuse count is the

number of accesses a block gets before it is evicted. Prior research shows that the reuse distance and reuse count can be predicted dynamically to improve CPU cache replacement algorithms. In an optimal case, a predictor can learn the reuse behavior from the history of each memory block. However, keeping track of every block requires impractical hardware storage overhead. It has been observed in prior CPU cache studies that cache blocks can be grouped together so that each group of cache blocks shares similar characteristics. For example, the fetching PC, the last touch PC and the memory region address have been proven to be good grouping criteria. Furthermore, the PC or memory region address can be hashed to generate a signature for the group and thus the total storage size of the history table can be reduced significantly. The signature-based reuse predictor requires the cache blocks with the same signature to share a similar reuse behavior. The new predictor generates the signature for a memory request by hashing the PC and the relative age of the fetching warp. A signature generated by hashing both the PC and the relative age of the warp can identify the memory accesses based on their scheduler priority and the PC itself.

The basic function of the predictor is to maintain the reuse distance and reuse count history for each memory block signature and to predict the reuse distance for any newly fetched cache block. The predictor is implemented as a signature indexed reuse-history table. It consists of two tables: the signature table and the reuse history table. Each entry in the signature table stores an existing signature and the signature is used to index the reuse-history table.

Each entry in the reuse history table summarizes the reuse information for all existing memory blocks that share the corresponding signature. Each entry of the reuse information table consists of five fields: a sum-of-reuse-distance field, a total-of-reuse-count field, a total-fetching-count field, an average-reuse-count field and an average-reuse-distance field.

The major operations of the reuse distance predictor are summarized as follows.

1. Update

   The sampler sends update requests to update the corresponding entries in the reuse history table. There are two kinds of update request: hit update and miss update. Each request integrates a type-ID to identify itself as a hit update or a miss update, and a signature of the new request address. Additionally, a hit update includes the reuse distance of the cache hit. A miss update does not need extra information.

   The signature table performs an associative search to check whether the signature exists in the table already. If the entry does not exist, one entry in the signature table is allocated and initialized to the new signature.

   If the signature is already in the table, a hit update adds the reuse distance value to the sum of reuse distance field. The sum of reuse count field is also increased by one. A miss update increases the total fetching number by one.

   If either of these three fields is saturated, the average reuse distance

field and the average reuse count field are updated with the new values calculated as follows.

$$average{-}reuse{-}distance = sum{-}of{-}reuse{-}distance/total{-}reuse{-}count; \tag{6.1}$$

$$average{-}reuse{-}count = total{-}reuse{-}count/total{-}fetching{-}count; \tag{6.2}$$

After updating the average-reuse-distance and average-reuse-count values, the other three fields are reset to zero.

2. Prediction

When a new block is fetched into the cache, the cache controller sends the signature to the reuse predictor to get the predicted reuse distance and count. The predictor searches the incoming signature in the signature table. If there is no match, the predictor returns zero to the cache controller, which means no prediction can be made for this signature. Otherwise, the predictor reads the corresponding reuse history and sends the average reuse distance and average reuse count to the cache controller.

6.2.2 Cache Block Reuse Distance Sampler

The reuse-prediction-based replacement approach relies on a sampler to collect the reuse distance and the signature on a cache hit and the signature on a cache miss. As we discuss in Section 6.2.1, the sampler sends two kinds of update to the predictor: the hit update and the miss update.

A full size sampler can be considered as a duplication of the tag array, which is organized the same as the tag array in the L1 cache. The sampler does not store the cache block data, instead, each tag is associated with an entry, which stores the signature of the correspond cache block and a counter field to calculate the reuse distance. Although the full size sampler is able to provide the reuse distance of all cache hits, it requires impractical hardware costs and extra power consumption. Fortunately, prior research shows that the sampler does not need to duplicate the tag array for all cache sets. In fact, a sampler that duplicates the tag array of 1/32 of all cache sets has been proven to be sufficient.

The major operations of the sampler are summarized as follows.

1. Initialization

   When a new cache block is fetched to a set that the sampler is monitoring, the sampler stores the signature value, and initializes the counter value to zero.

2. Counter Update

   Each access to the cache sets that are under sampler's monitoring will trigger the sampler to increase the counter for the set by one.

3. Hit update

   On a cache hit, for the tag entry that matches the new access, the difference between the value of the counter for set and the last-access-counter

value of the entry is considered as the reuse distance of the stored signature. The sampler sends the signature and the reuse distance value as a hit update to the reuse distance predictor.

4. Miss update

   On a cache miss, the signature of the incoming request is sent to the reuse predictor as a miss update.

6.2.3 Cache Replacement Controller

The new cache replacement controller includes two parts: the per-cache line reuse-distance field and the replacement decision unit. The controller works as follows.

1. Initialization

   When a new cache line is fetched into the cache, the controller first calculates the signature based on the new block's fetching PC and the relative age of the warp that issues the request. This information has been attached to the memory request packet. The signature is sent to the reuse predictor, which then sends back the predicted reuse distance and reuse count value. The reuse distance and count value then are stored together with the cache line.

2. Updates at access

   When a new request accesses the cache set, all the cache blocks in the

set update their reuse distance field by decreasing the value by one to keep track of the distance of the future reuse.

3. Updates at hit

   When a new request hits a cache block, the block updates its reuse count field by decreasing the value by one to indicate the number of reuses left.

4. Replacement victim selection

   In this research, we keep three values with each cache line: the reuse count, the reuse distance and the ID of the warp fetching the block. The ID of the warp can be used to calculate the age of the fetching warp as we discuss in Section 5.2. The score of each cache line can be calculated based on these three values with different functions, such as reuse distance only, age-only functions and counter-based.

   When the warp that fetches the block is no longer active, or the reuse count value of a block reaches zero, the block can be considered as dead. The replacement controller selects the blocks that are predicted to be dead over the other blocks in the same cache set. If no dead block is found, the controller calculates the score of all cache blocks in the set. It first selects the blocks with the lowest score. If there is more than one cache block sharing the lowest score, the least recently used block is evicted.

6.2.4 Cache Block Scoring Strategy

The reuse-prediction-based replacement approach (RPR), keeps the predicted reuse information together with the cache line including the reuse distance, the reuse count, and the active warp ID (AWID) of the warp fetching the block. The AWID can be applied to calculate the age of the fetching warp which represents the scheduler priority of the warp.

We propose to calculate a score for each of the cache blocks based on the three inputs: reuse distance (RD), reuse count (RC) and age of the warp (AGE). The block with the lowest score is selected as the replacement victim. With different strategies to calculate the score, the RPR mechanism can approximate several replacement policies.

AgeLRU algorithm: We propose the AgeLRU algorithm in Chapter 5. The RPR mechanism can be considered as a mechanism that augments the AgeLRU mechanism with reuse distance and reuse count prediction. If the cache line score is calculated without considering the reuse distance and the reuse count of the block, the RPR algorithm works in the same way as the AgeLRU algorithm. The score function for AgeLRU algorithm is

$$Score_{AgeLRU} = 256 - AGE. \tag{6.3}$$

The relative age of the oldest warp is zero and the $Score_{AgeLRU}$ function enables the oldest warp to have the highest score.

Reuse-Distance-based Replacement Algorithm: The reused-distance-base replacement algorithm [42] is one of the CPU cache pollution-resistant algorithms. It hashes the PC to generate the signature which is used to categorize the memory references. However, in the case of GPUs, the instructions that share the same PC but are in warps with different ages show very different reuse behavior because the thread scheduler assigns them different priorities. Therefore, the RPR mechanism approximates this algorithm on the GPU cache while hashing both the PC and the age of the warp to generate the signature.

The score function for reuse-distance-based algorithm is

$$Score_{reuse-dist} = 256 - RD. \tag{6.4}$$

Improved counter-based cache replacement algorithm: The counter-based replacement algorithm [44] is a dead-block prediction algorithm. The reuse count of a block is predicted. When the access count of the block exceeds the threshold, the block is identified as dead and thus gets evicted. The RPR mechanism also predicts the reuse count of each block and decreases the reuse count value by one at each cache hit. RPR can approximate the counter-based dead block prediction algorithm. However, this algorithm mostly aims to improve the efficiency of high-associative caches. The associativity of GPU L1 cache is generally low. For instance, our baseline GPU L1 cache is a four-way associative cache.

We augment the counter-based cache replacement algorithm with the

ability to address the cache pollution problem. The algorithm allows a block with more reuse left to evict a block with less reuse left. The score function of the improved counter-based algorithm is

$$\begin{aligned} If\,(RC > 0)\, Score_{counter-based} &= 256 + RC; \\ Else\, Score_{counter-based} &= 0; \end{aligned} \tag{6.5}$$

6.3 Optimizing Hardware Cost

The implementation cost of the reuse-prediction-based replacement approach mainly consists of three parts: the reuse history table, the reuse sampler, and the per-cache line reuse storage. In this section, we describe the optimizations to reduce the hardware cost.

History Table in the Predictor: The reuse distance/count predictor relies on the history to keep track of the reuse information of each memory block group. The total number of entries in the table depends on how many active signatures are present during execution. The table is shared among all SMs. In our experiment, the table can keep up to 512 entries. Each entry of the table consists of five fields: a 11-bit sum-of-reuse-distance field, a 8-bit total-of-reuse-count field, a 5-bit total-fetching-count field, a 4-bit average-reuse-count field and a 4-bit average-reuse-distance field. The total size of the table is 2K bytes.

Shadow Tag Array in the Sampler: The full size sampler keeps track of the reuse distance and reuse count of all cache blocks with a duplication of the tag array (i.e. a shadow tag array). The hardware cost of the shadow tag array makes the approach impractical. Fortunately, prior research observes that the sampler only needs to monitor a small subset of the cache sets to learn the reuse behaviour of all cache blocks. We apply a similar technique in our implementation where the sampler only keeps track of 1/4 of the cache sets. For each set that the sampler monitors, there is an 2-byte access counter. For each block in the set, the sampler consists of a 3-byte tag, 2-byte last access counter value, and 1-byte signature. For a 4-way 16KB L1 cache with 32 sets, the size of the sampler is 208 bytes. The sampler only needs to be implemented on one SM.

Per-cache-block meta data: In this research, we keep three fields of meta data with each cache line: 4-bit reuse count, 4-bit reuse distance, and 8-bit ID of the warp fetching the block. The meta data size for the 16KB L1 cache is 256 Bytes.

In summary, the hardware cost of implementing the RPR approach includes the 2K-byte reuse history table, the 208-byte sampler shared among all SMs and the 256-byte per-SM meta data storage. For our baseline GPU configured as in Table 3.4, the total storage overhead is 6096 bytes.

RPR can approximate multiple algorithms with various cache block score functions. If the results show that there is no performance benefit for

approximating some of the algorithms, the related reuse information does not need to be stored.

6.4 Results

In this section we present results for the reuse-prediction-based cache replacement algorithms.

We configure the RPR mechanism with three cache line score strategies as we discuss in Section 6.2.4 to approximate the AgeLRU algorithm, reuse-distance algorithm and improved counter-based algorithm. The results are compared to static CTA-throttling (`CTA-opt`) in our experiments. The experimental results show that the reuse-prediction-based replacement enables a throughput improvement of 5% on average for the regular applications and a speedup of 3.2% across all the key benchmarks over the AgeLRU algorithm. At the end, we summarize the results and make the conclusions of this chapter.

Figure 6.3 summarizes the L1 miss rate of the best performing static CTA throttling, the counter-based replacement, the reuse-distance-based replacement and the AgeLRU replacement algorithms.

As shown in Figure 6.3, static CTA throttling can reduce the L1 miss rate significantly (12% on average) and it achieves a speedup of 47% on average. The benefit is mostly from solving the cache thrashing problem. Our approach is implemented on top of the CTA-throttling approach to address the cache pollution problem.

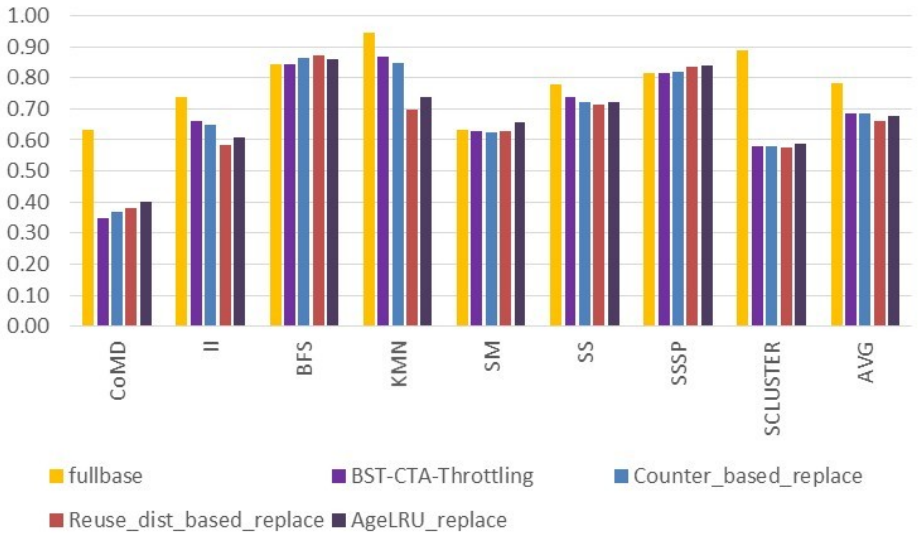


Figure 6.3: L1 miss rate of CTA Throttling, reuse-distance-based replacement, reuse-counter based replacement and AgeLRU replacement (normalized to baseline where maximum number of CTA is allowed.)

Compared to the CTA-throttling approach, the three algorithms that RPR approximates, namely the counter-based replacement, the reuse-distance-based replacement and the AgeLRU algorithm, are able to reduce the cache miss rate for II, SS and KMN. On the other hand, all three algorithms increase the L1 miss rate for CoMD, BFS and SSSP. The L1 miss rate of the other two benchmarks, SCLUSTER and SM, are not sensitive to these algorithms. Among these three algorithms, the reuse-distance-based replacement algorithm achieves the largest L1 miss rate reduction for most of the benchmarks. It reduces the L1 miss rate for II, KMN and SS by 8% 17% and 3% respectively.

Figure 6.4 summarizes the performance of the best performing static CTA throttling, the counter-based replacement, the reuse-distance-based re-

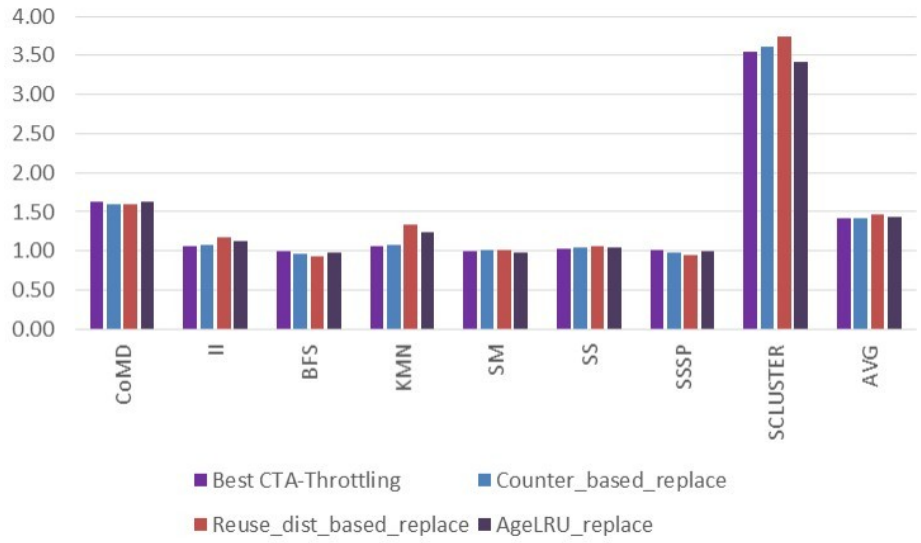


Figure 6.4: Speedup of CTAThrottling, reuse-distance-based replacement, reuse-counter based replacement and AgeLRU replacement (normalized to baseline where maximum number of CTA is allowed.)

placement and the AgeLRU replacement algorithms. As shown in this figure, on average, the reuse-distance-based replacement algorithm performs better than the other algorithms. It outperforms CTA throttling, counter-based replacement and AgeLRU replacement by 6%, 6% and 5% respectively on average across all the key benchmarks. For most of the benchmarks, the speedup it achieves exhibits a strong correlation with the L1 miss rate reduction it achieves.

SSSP and BFS are the outliers in our experiment, on which the reuse-distance-prediction-based replacement shows negative effect. This algorithm degrades the performance by 7% and 5% respectively. The reason for this negative effect is that BFS and SSSP are are known irregular GPU programs

which travel irregular graphs. They are both from the LonestarGPU benchmark set [7], which is a collections of GPU applications showing irregular behavior. The cache behavior of each thread depends on the number of connections that each node has in this graph. Our approach trains the prediction table based on reuse patterns in the history. Unfortunately, these history reuse patterns are not necessarily repeating due to the irregularity of the graph.

In summary, the reuse-prediction-based replacement is able to achieve 6% speedup over the optimal CTA throttling mechanisms across the eight key benchmarks we test. For the six regular applications, we observe a L1 miss rate reduction of 4% and a speedup of 10% over the optimal CTA-throttling mechanism. Compared to the AgeLRU algorithm we propose in Chapter 5, the RPR algorithm enables a throughput improvement of 5% on average for the regular applications and a speedup of 3.2% across all the key benchmarks.

6.5 Summary

In this chapter, we propose and evaluate a new reuse-prediction-based GPU L1 cache replacement mechanism (RPR) to address the intra-thread cache pollution problem. This mechanism predicts the reuse distance and reuse count of each memory block and applies the reuse information to improve the replacement decision.

CPU cache-pollution-resistant algorithms can not be applied directly to a GPU L1 cache. RPR leverages the GPU thread scheduling priority to generate the signature that indexes the prediction table. It also calculates the

average reuse distance of each memory request group to tolerate unpredictable thread interleaving.

RPR can approximate the reuse-distance-based algorithm, the counter-based algorithm, and the AgeLRU algorithm with various cache block score functions. Among these three configurations, the reuse-distance-based algorithm outperforms the others. Compared to the AgeLRU algorithm we propose in Section 5, the reuse-distance-based algorithm enables a throughput improvement of 5% on average for the regular applications and a speedup of 3.2% across all the key benchmarks.

Our experimental results show that the reuse-distance-based algorithm degrades the performance of the irregular GPU applications. In these applications, the reuse histories are not necessarily repeating due to the irregularity of the program. For instance, SSSP computes the shortest path from a source node to all nodes in a graph. The cache behavior of each thread depends on the number of connections that each node has in this graph. Our approach trains the prediction table based on reuse patterns of the nodes that have been evaluated in the history. Unfortunately, these reuse patterns are not necessarily repeating on other nodes in the graph.

We believe a dynamic RPR mechanism that selects the RPR algorithm and the LRU algorithm adaptively, could avoid hurting the irregular applications. The dynamic RPR mechanism can be implemented with the parallel voting mechanism similar to the Dynamic-AgeLRU algorithm we discuss in Section 5.3. We leave the dynamic-RPR as a part of future work.

Chapter 7

Conclusion

While massively threaded processors such as GPUs are able to provide high throughput, implementing a high throughput memory system is challenging. The gap between the peak arithmetic capability and the off-chip bandwidth will inevitably grow in the future. At the same time, exploiting locality in these systems can be difficult because of the competition for cache capacity by the threads. This dissertation advocates orchestrating thread scheduling with cache management polices to improve cache efficiency and memory system throughput. Based on this principle, we propose three mechanisms including Priority-based Cache Allocation mechanism (PCAL), AgeLRU and Reuse-Prediction based Replacement (RPR) algorithms to improve the memory system throughput. PCAL demonstrates that the thread scheduler needs to consider the effect of Thread Level Parallelism (TLP) on the cache performance and memory resource usage to maximize the memory throughput. By monitoring memory system statistics, PCAL explicitly determines the number of threads that share the cache and the minimum number of bypassing threads that saturate memory system resources. This approach reduces the cache thrashing problem and effectively employs memory system resources that would otherwise go unused by a pure thread throttling approach. The

AgeLRU and RPR algorithms also follow the principle of our thesis statement by considering the effect of the thread scheduling algorithm. These algorithms are able to outperform the thrashing-resistant and pollution-resistant algorithms adapted from CPU variants. We expect the three techniques might combine synergistically to further improve the throughput.

7.1 Dissertation Contributions

There are four major contributions of this work:

1. **Investigating the memory system effect of high TLP and identifying the performance bottleneck of each application**

   Mapping the maximum amount of parallel work that the hardware can support often leads to cache thrashing and saturation of memory system resources, which does not necessarily ensure the best overall performance. To investigate how the number of active hardware threads affects GPU performance, the cache hit ratio and other chip resource utilization metrics, we characterized a set of cache-sensitive applications by varying the maximum number of warps that each thread scheduler allows. We analyzed the Instruction Per Cycle (IPC), L1/L2 cache miss ratio and the chip resource utilization metrics including the memory request round trip latency, the Network-on-Chip (NoC) transmission latency and the DRAM bandwidth utilization. We identified the major performance bottlenecks for each application when TLP is increased. Applications are grouped into categories based on these bottlenecks.

Based on the bottleneck identification results, we observed that throttling techniques rely on tuning only one parameter — the total number of threads — to make a trade-off among the total TLP, the cache miss ratio and different resources in the memory system. The trade-off is often sub-optimal.

We observed two opportunities to utilize the memory system more effectively. First, when chip resources, such as off-chip bandwidth or NoC bandwidth etc., become saturated, we showed that cache-sensitive workloads have the opportunity to increase cache locality, but not merely by increasing cache size. The key elements to enable better usage of the cache resources without sacrificing overall parallelism are: (1) reducing the threads that compete for the cache, and (2) maintaining the total TLP by allowing other threads to bypass the cache. Second, when the total TLP has to be compromised to maintain an acceptable cache hit ratio to maximize the overall throughput, we demonstrated that adding bypassing threads to utilize the spare resources can effectively improve the throughput.

2. **Addressing cache thrashing and memory system resource saturation with a thread-scheduling directed cache allocation mechanism**

   We enhanced the thread throttling technique with the Priority-based Cache Allocation mechanism (PCAL) to address inter-thread L1 cache thrashing and memory system resource saturation. Unlike thread throt-

tling approaches which force all threads to feed the L1 cache, PCAL divides threads into three subsets: (1) threads that issue and share the cache, (2) threads that issue but bypass one or more levels of cache, and (3) threads that are throttled and do not issue. PCAL can improve performance with two optimization strategies: either increasing TLP while maintaining cache hit ratio, or optimizing cache hit ratio while maintaining TLP.

We developed dynamic PCAL, which not only determines the best strategy to improve performance, but also decides the size of the cache thread group and the bypassing thread group. Dynamic PCAL monitors the chip resource usage to decide whether there are sufficient resources to support a bypassing thread. This approach reduces cache thrashing and effectively employs the chip resources that would otherwise go unused by a pure thread throttling approach. We observed on average 67% improvement over the original as-is benchmarks and a 18% improvement over a better-tuned warp-throttling baseline.

3. **Developing the AgeLRU and Dynamic-AgeLRU GPU-specific thrashing-resistant cache replacement and bypassing algorithms** We investigated the feasibility of applying a set of thrashing-resistant CPU cache algorithms to the GPU cache hierarchy, including the Bimodal Insertion Policy (BIP) mechanism [71] and the Dynamic Insertion Policy (DIP) mechanism [71]. We observed that there are two problems that limit the feasibility of adapting these CPU cache algorithms directly

to the GPU L1 cache. (1) BIP randomly selects memory blocks to reside in the cache. It tends to activate all warps concurrently and thus it increases the total working set size. (2) DIP relies on the set-dueling mechanism to evaluate two algorithms on different sets of the same cache. In a GPU, the thread interleaving order and cache access pattern may change when set-dueling is applied to estimate two algorithms.

We proposed the AgeLRU and Dynamic-AgeLRU mechanisms, which are thread scheduling-aware replacement and bypassing algorithms to overcome the inter-thread thrashing problem. When selecting a collection of memory blocks to reside in the cache, AgeLRU reduces the number of warps that share the cache-resident blocks by prioritizing older warps. Dynamic-AgeLRU selects the AgeLRU or the LRU algorithm adaptively based on a parallel voting mechanism.

Compared to the LRU algorithm, the three AgeLRU algorithms — replacement, bypassing and bypassing with traffic optimization-enabled — increase performance by 4%, 8% and 28% respectively across fourteen cache-sensitive benchmarks. Our results show that Dynamic-AgeLRU algorithms can avoid degrading the performance of non-thrashing applications by selecting the LRU algorithm. We also demonstrated that the Dynamic-AgeLRU algorithms are able to achieve the majority of the performance improvements that the AgeLRU algorithms can achieve for most applications.

4. **Proposing Reuse-Prediction based Cache Replacement Algo-**

rithm (RPR)

It is desirable to apply CPU cache pollution-resistant algorithms to the GPU cache to address pollution. However, we observed that without considering the effects of the thread scheduling, highly accurate reuse prediction is not achievable.

We developed a Reuse-Prediction-based cache Replacement scheme (RPR) for a GPU L1 data cache to address the intra-thread cache pollution problem. This scheme identifies and prioritizes the near-reuse blocks and high-reuse blocks to maximize the cache efficiency so that it can achieve further throughput improvement beyond the AgeLRU mechanism. RPR uses the GPU thread scheduling priority to generate a signature that indexes the prediction table. RPR can approximate the reuse-distance-based algorithm, the counter-based algorithm, and the AgeLRU algorithm with various cache block score functions. Among these three configurations, our results show that the reuse-distance-based algorithm outperforms the others. Compared to the AgeLRU algorithm we proposed in Section 5, the reuse-distance-based algorithm enables a throughput improvement of 5% on average for regular applications, and a speedup of 3.2% across a set of cache-sensitive benchmarks.

7.2 Future Work

Increasing TLP and optimizing cache hit ratio synergistically

Section 4.3.1 introduces the two performance optimization strategies that the

PCAL approach exploits: **(A) increasing TLP while maintaining cache hit ratio, and (B) increasing cache hit ratio while maintaining TLP.** In our current implementation, PCAL applies one of the two strategies to every new program phase. However these two optimization strategies can work synergistically to further improve the overall throughput.

For a resource constrained application, we can apply strategy B first to improve the hit ratio while maintaining the TLP. The number of cache threads is reduced to optimize the overall cache hit ratio. As we show in Figure 4.18, this strategy enables applications to achieve higher cache hit ratios. As a result, the number of outstanding memory requests is reduced, and the GPU memory system is less stressed. For instance, as shown in Figure 4.20, the NoC latency and the round-trip latency of CoMD both decrease. Since the memory system is less saturated or possibly even under-utilized, the application can become a non-resource constrained application. There is a further opportunity to apply strategy A to increase TLP to utilize the spare resources.

For a non-resource constrained application, we can apply strategy A first to increase TLP by enabling extra bypassing threads until memory system resources saturate. After this step, the application turns into a resource constrained application. There is a further opportunity to apply strategy B to rebalance the number of cache threads and the number of bypassing threads to maximize the cache hit ratio and overall performance. Future work should study the interaction between the two optimization strategies to increase the TLP and optimize overall cache hit ratio at same time.

Allocating L1 and L2 cache to different threads group explicitly

As discussed in Section 4.8.4, PCAL only focuses on the L1 cache allocation and is not able to achieve further speedup when it is applied to the L2 cache. Figure 4.26 shows that applying PCAL to the L2 in addition to the L1 leads to performance degradations compared to PCAL on L1 only.

There are two problems that limit the performance of the current implementation of PCAL on the L1 and L2 mechanism: (1) The current implementation of PCAL assigns L1 tokens and L2 tokens to the same subset of threads. However, the token number that maximizes the L1 cache efficiency is not necessarily the token number that can maximize the L2 cache efficiency. (2) The L1 token-holder threads can fit their working sets in the L1 cache. The L1 cache misses of the loads from these threads are mostly compulsory misses. Assigning these requests high priority for L2 allocation does not help them hit in the L2 cache. Moreover, it reduces the opportunity of the non-token-holder threads to reside in the L2 cache and thus degrades the L2 hit ratio of these threads.

We believe the GPU throughput can be further improved by allocating L1 and L2 cache capacity separately to allow each cache level to be utilized more effectively. Such a scheme could allocate the L1 and L2 cache capacity separately to two subsets of threads. The two subsets could be either inclusive or exclusive to balance the L1 and L2 hit ratio. Both the L1 token number and L2 token number can be selected adaptively according to program resource characteristics and hardware resource availability. Consequently, the working

sets of the two thread groups can reside in different levels of caches without trashing either of them. This scheme could reduce the L2 miss ratio and further improve the performance over the current PCAL scheme.

Combining PCAL, AgeLRU and RPR

The three mechanisms we propose in this work all aim to improve the overall GPU memory system throughput. They could potentially work together synergistically to further improve throughput. When the PCAL mechanism is enabled, threads are divided into three subsets: (1) threads that issue and share the cache, (2) threads that issue but bypass one or more levels of cache, and (3) threads that are throttled and do not issue. PCAL does not require a specific cache replacement algorithm. AgeLRU and RPR algorithm could be applied together with PCAL to further improve the performance. (1) AgeLRU can be applied to manage the memory request stream from the cache threads. When PCAL determines the number of cache threads, it aims to maximize the overall cache hit ratio. Consequently, the cache threads may not maintain their full working sets in the cache. These threads could still compete for cache capacity. PCAL does not assign priorities among these threads. AgeLRU can be applied to manage their working set by prioritizing the older ones at cache replacement. Consequently, the thrashing among cache threads can be partially avoided. (2) PCAL assigns the cache allocation priority (i.e. token) by warp. This only helps alleviate the inter-thread cache thrashing problem. Both the low-reuse blocks and the high-reuse blocks are treated equally. The

low-reuse blocks of the cache threads can reside in the cache until the warp finishes. This wastes the limited cache capacity and degrades cache efficiency. RPR is able to identify the high-reuse blocks by predicting their reuse behavior. It can be applied to filter out the low-reuse blocks from the cache threads and thus each cache thread occupies less cache. Consequently, more threads can be allowed to share the cache.

Combining PCAL, AgeLRU and RPR could address the inter-thread thrashing, intra-thread pollution and memory system resource saturation all together. The combined technique may better utilize the cache capacity and other memory system resources more effectively improving the GPU throughput beyond what was shown in this dissertation.

7.3 Practicality Discussion

To improve the memory system throughput, this dissertation proposes three mechanisms: Priority-based Cache Allocation mechanism (PCAL), AgeLRU, and Reuse-Prediction based Replacement (RPR) algorithms. These mechanisms involve both programmer interventions and architecture innovations to further exploit data locality and utilize the on-chip and off-chip resources more effectively. It is important to reduce hardware cost and programmer effort to enable our new mechanisms to be practical for future throughput processors. This section discusses implementation options and the practicality of these three mechanisms.

Priority-based Cache Allocation

This dissertation proposes the PCAL mechanism as an enhancement of throttling techniques and provides two implementation options: static PCAL and dynamic PCAL. Static PCAL achieves a higher throughput than dynamic PCAL with a lower hardware cost. But it needs more programmer intervention than dynamic PCAL. The implementation of static PCAL is relatively simple. The hardware cost includes token assignment logic within the thread scheduler and a one-bit priority field for each cache line. Static PCAL requires programmers to provide the number of extra bypassing threads in addition to the optimal number of threads that is required by pure throttling techniques. Static PCAL matches the needs of experienced programmers. By allowing the programmers to determine the value of the two parameters for each kernel, static PCAL provides an effective tool to tune the cache efficiency and the memory system throughput of throughput processors.

Dynamic PCAL determines the number of bypassing threads and the number of cache threads dynamically to ease the burden of normal programmers. It does not require programmers to have a deep understanding of GPU microarchitectures. This dissertation implements dynamic PCAL as a pure hardware mechanism. The algorithms to search the best configurations, including hill-climbing and parallel voting, are implemented as a hardware state-machine. The hardware cost of the state machine is not high, but the pure hardware based dynamic PCAL does have two disadvantages: (1) it increases the design complexity of the thread scheduler, and (2) the hardware imple-

mentation of the algorithms limit its flexibility. More complex algorithms may be impractical to implement in hardware.

We observe that implementing parts of the dynamic PCAL mechanism in the runtime software can reduce the design complexity and improve the flexibility of the mechanism effectively. In the runtime/hardware combined implementation of PCAL, the hardware part is the same as in static PCAL. It reads the PCAL parameters from registers. Unlike static PCAL, which requires the programmer to set the value of the PCAL parameters, the new implementation relies on the runtime to monitor the memory resource usage and run the hill-climbing and parallel voting algorithm to determine the PCAL parameters.

The runtime/hardware combined PCAL requires a hardware cost as low as static PCAL. The configuration searching algorithms are implemented as runtime functions, which is flexible, creating new opportunities to design new algorithms without increasing hardware cost. With this implementation, static PCAL and dynamic PCAL can be combined under the same framework. The programmers are still able to call a runtime function to set the PCAL configurations directly. When the runtime loads a new kernel, it first checks whether the PCAL parameters have been set by the software. If the parameters have been set, it allows the hardware to read the value directly from the registers. If the parameters have not been set by programmer, the runtime runs the algorithm to determine the configuration parameters of PCAL. Consequently, the runtime/hardware combined PCAL becomes an effective

and practical mechanism with low hardware overhead and high flexibility. We believe it fits the needs of both the experienced programmers and the normal programmers.

Scheduling-aware GPU cache algorithms

The GPU thread scheduler shapes memory access patterns directly and thus has a significant effect on cache efficiency. Based on this observation, this dissertation proposes several scheduling-aware GPU cache replacement and bypassing algorithms including AgeLRU, Dynamic-AgeLRU and Reuse-Prediction-based Replacement (RPR). Among these algorithms, AgeLRU and Dynamic-AgeLRU are proposed to address the inter-thread cache thrashing problem, achieving speedups of 28% and 20% respectively. The hardware cost of implementing AgeLRU includes an 8-bit active warp ID field in each cache line and the changes in the replacement logic. For a 16KB GPU L1 cache, the storage cost is about 128 bytes, which is less than 1% of the cache. Dynamic-AgeLRU augments the AgeLRU algorithm with a parallel voting mechanism, which selects between the AgeLRU algorithm and the LRU algorithm adaptively to avoid degrading the performance of non-thrashing applications. The parallel voting mechanism is implemented as a 3-state state machine, which only requires simple logics. The significant speedup and the low hardware cost make the Dynamic-AgeLRU mechanism practical. We believe it is a good candidate to replace the LRU algorithm for a GPU cache.

The Reuse-Prediction-based cache replacement algorithm aims to pre-

dict the reuse behavior of each cache block, identify the near-reuse blocks and improve the cache replacement decision. However, it only achieves an average speedup of 3.5% beyond the AgeLRU algorithm. Reasons are two-fold:

- For irregular applications, the reuse predictor can not predict the reuse behavior accurately. RPR trains the prediction table based on reuse patterns in the history. Unfortunately, these history reuse patterns are not necessarily repeating due to the irregularity of the program.

- The fine-grained thread interleaving degrades the prediction accuracy. The thread interleaving order is directly affected by the thread scheduler. Although RPR considers the scheduling priority of the fetching thread for the cache block when it predicts the reuse behavior, the reuse distance of the cache blocks sharing a signature sometime diverge significantly. Furthermore, some unexpected long latency operations like shared memory bank conflicts could also change the thread interleaving order, which is often unpredictable.

The RPR scheme is not able to predict the cache block reuse behavior accurately to achieve a good speedup because of the effect of the fine-grained thread interleaving. Therefore we do not feel it is a practical design option to implement RPR to replace the Dynamic-AgeLRU algorithm.

7.4 Concluding Thoughts

Throughput processors must continue to improve memory system throughput to maintain high occupancy for the computation units. By orchestrating thread scheduling with cache management polices, this dissertation proposes several mechanisms that improve cache efficiency and memory system throughput significantly. The two most practical mechanisms, PCAL and AgeLRU, both make cache management decisions at the warp level. To further improve cache efficiency, it is desirable to identify and prioritize the near-reuse blocks and high-reuse blocks to maximize cache efficiency and achieve further throughput improvements beyond warp level cache management algorithms. However, the reuse prediction based replacement algorithm study demonstrates the difficulty in predicting the reuse behavior of cache blocks. It opens up two broad challenges for future work: (1) overcoming the effect of thread interleaving to predict the reuse behavior of GPU cache blocks accurately, and (2) understanding the reuse patterns of the irregular applications. We envision that an accurate predictor for cache block reuse will be the key element that further improves GPU memory system throughput in the future.

Bibliography

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

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

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

[4] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2009.

[5] Kristof Beyls and Erik D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, IASTED, Anaheim, California, USA, 2001*, pages 617–622, 2001.

[6] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd annual international symposium on Computer architecture*, ISCA '96, pages 78–89, New York, NY, USA, 1996. ACM.

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

[8] Henk Corporaal Cedric Nugteren, Gert-Jan van den Braak and Henri Bal. A detailed gpu cache model based on reuse distance theory.

[9] Jichuan Chang and Gurindar S Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 242–252. ACM, 2007.

[10] 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*, October 2009.

[11] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 13:1–13:11, New York, NY, USA, 2011. ACM.

[12] Hsiang-Yun Cheng, Chung-Hsiang Lin, Jian Li, and Chia-Lin Yang. Memory latency reduction via thread throttling. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 53 –64, dec. 2010.

[13] Jamison D. Collins and Dean M. Tullsen. Hardware identification of cache conflict misses. In *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 126–135, Washington, DC, USA, 1999. IEEE Computer Society.

[14] Nam Duong, Dali Zhao, Taesu Kim, R. Cammarota, M. Valero, and A.V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 389–400, Dec 2012.

[15] A. Duran and M. Klemm. The intel many integrated core architecture. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 365 –366, july 2012.

[16] 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.

[17] 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.

[18] 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.

[19] Antonio González, Carlos Aliagas, and Mateo Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pages 338–347, New York, NY, USA, 1995. ACM.

[20] S. Grauer-Gray, Lifan Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, May 2012.

[21] Z. Guz, O. Itzhak, I. Keidar, A. Kolodny, A. Mendelson, and U.C. Weiser. Threads vs. caches: Modeling the behavior of parallel workloads. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 274–281, Oct 2010.

[22] Zvika Guz, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C. Weiser. Many-Core vs. Many-Thread Machines: Stay Away From the Valley. *IEEE Computer Architecture Letters*, Jan 2009.

[23] 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.

[24] Enric Herrero, José González, and Ramon Canal. Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors. *ACM SIGARCH Computer Architecture News*, 38(3):419–428, 2010.

[25] C.J. Hughes, Changkyu Kim, and Yen-Kuang Chen. Performance and energy implications of many-core caches for throughput computing. *Micro, IEEE*, 30(6):25–35, Nov 2010.

[26] Jaehyuk Huh, Doug Burger, and Stephen W. Keckler. Exploring the design space of future cmps. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 199–210, Washington, DC, USA, 2001. IEEE Computer Society.

[27] INTEL. Inside the Intel Itanium 2 Processor, July 2002.

[28] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 25–36. ACM, 2007.

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

[30] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely Jr, and Joel Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 208–219. ACM, 2008.

[31] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[32] J. Jalminger and P. Stenstrom. A novel approach to cache block reuse predictions. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 294–302, Oct 2003.

[33] 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.

[34] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. Mrpb: Memory request prioritization for massively parallel processors. In *20th International Symposium on High Performance Computer Architecture (HPCA-20)*, 2014.

[35] Yunlian Jiang, Eddy Z Zhang, Kai Tian, and Xipeng Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Compiler Construction*, pages 264–282. Springer, 2010.

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

[37] 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.

[38] Teresa L. Johnson, Daniel A. Connors, Matthew C. Merten, and Wen-mei W. Hwu. Run-time cache bypassing. *IEEE Trans. Comput.*, 48(12):1338–1354, December 1999.

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

[40] S.W. Keckler, W.J. Dally, B. Khailany, M. Garland, and D. Glasco. Gpus and the future of parallel computing. *Micro, IEEE*, 31(5):7 –17, sept.-oct. 2011.

[41] Kamil Kedzierski, Miquel Moreto, Francisco J Cazorla, and Mateo Valero. Adapting cache partitioning algorithms to pseudo-lru replacement policies. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[42] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pages 245–250, Oct 2007.

[43] M. Kharbutli and D. Solihin. Counter-based cache replacement and bypassing algorithms. *Computers, IEEE Transactions on*, 57(4):433–447, April 2008.

[44] M. Kharbutli and Yan Solihin. Counter-based cache replacement algorithms. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 61–68, Oct 2005.

[45] An-Chow Lai, C. Fide, and B. Falsafi. Dead-block prediction amp; dead-block correlating prefetchers. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 144–154, 2001.

[46] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 144–154, New York, NY, USA, 2001. ACM.

[47] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.*, 50(12):1352–1361, December 2001.

[48] Jaekyu Lee, Nagesh B. Lakshminarayana, Hyesoon Kim, and Richard Vuduc. Many-thread aware prefetching mechanisms for gpgpu applications. In *Proceedings of the 2010 43rd Annual IEEE/ACM International*

Symposium on Microarchitecture, MICRO '43, pages 213–224, Washington, DC, USA, 2010. IEEE Computer Society.

[49] Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. Thread tailor: Dynamically weaving threads together for efficient, adaptive parallel applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 270–279, New York, NY, USA, 2010. ACM.

[50] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[51] W. Lin, S. Reinhardt, and D. Burger. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *7th International Symposium on High Performance Computer Architecture (HPCA-7)*, February 2001.

[52] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st Annual IEEE/ACM International*

Symposium on Microarchitecture, MICRO 41, pages 222–233, Washington, DC, USA, 2008. IEEE Computer Society.

[53] Yangchun Luo, Venkatesan Packirisamy, Wei-Chung Hsu, Antonia Zhai, Nikhil Mungre, and Ankit Tarkas. Dynamic performance tuning for speculative threads. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 462–473, New York, NY, USA, 2009. ACM.

[54] Nihar R Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck: problems and solutions. *Crossroads*, 5(3es):2, 1999.

[55] R. Manikantan, K. Rajan, and R. Govindarajan. Nucache: An efficient multicore cache organization based on next-use distance. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 243–253, Feb 2011.

[56] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[57] 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.

[58] Miquel Moreto, Francisco J Cazorla, Alex Ramirez, and Mateo Valero. Mlp-aware dynamic cache partitioning. In *High Performance Embedded*

Architectures and Compilers, pages 337–352. Springer, 2008.

[59] 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.

[60] Kyle J. Nesbit, James Laudon, and James E. Smith. Virtual private caches. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 57–68, New York, NY, USA, 2007. ACM.

[61] Alexandru Nicolau and Arun Kejariwal. How many threads to spawn during program multithreading? In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, LCPC'10, pages 166–183, Berlin, Heidelberg, 2011. Springer-Verlag.

[62] Jarek Nieplocha, Andrès Márquez, John Feo, Daniel Chavarría-Miranda, George Chin, Chad Scherrer, and Nathaniel Beagley. Evaluating the potential of multithreaded platforms for irregular scientific computations. In *Proceedings of the 4th International Conference on Computing Frontiers*, CF '07, pages 47–58, New York, NY, USA, 2007. ACM.

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

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

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

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

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

[68] NVIDIA Corporation. Tuning CUDA Applications for Kepler, July 2013.

[69] A.K. Parakh, M. Balakrishnan, and K. Paul. Performance estimation of gpus with cache. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2384–2393, May 2012.

[70] M. Qureshi and Y. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *39th International Symposium on Microarchitecture (MICRO-39)*, Dec 2006.

[71] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. *SIGARCH Comput. Archit. News*, 35(2):381–391, June 2007.

[72] 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.

[73] 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.

[74] 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.

[75] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. A locality-aware memory hierarchy for energy-efficient gpu architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 86–98, New York, NY, USA, 2013. ACM.

[76] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: challenges in and avenues for cmp scaling. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 371–382, New York, NY, USA, 2009. ACM.

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

[78] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM*

International Symposium on Microarchitecture, MICRO-46, pages 99–110, New York, NY, USA, 2013. ACM.

[79] Jennifer B. Sartor, Subramaniam Venkiteswaran, Kathryn S. McKinley, and Zhenlin Wang. Cooperative caching with keep-me and evict-me. In *Proceedings of the 9th Annual Workshop on Interaction Between Compilers and Computer Architectures*, INTERACT '05, pages 46–57, Washington, DC, USA, 2005. IEEE Computer Society.

[80] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-core x86 Architecture for Visual Computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008.

[81] Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 355–366, New York, NY, USA, 2012. ACM.

[82] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. Apogee: adaptive prefetching on gpus for energy efficiency. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 73–82. IEEE Press, 2013.

[83] 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. *IEEE Journal of Solid-State Circuits*, 46(1):131–144, 2011.

[84] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. Holistic run-time parallelism management for time and energy efficiency. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 337–348, New York, NY, USA, 2013. ACM.

[85] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and W-m Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[86] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 277–286, New York, NY, USA, 2008. ACM.

[87] SUN. UltraSPARC T2 supplement to the UltraSPARC architecture, 2007.

[88] I-Jui Sung, G.D. Liu, and W.-M.W. Hwu. Dl: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Computing (InPar), 2012*, pages 1–11, May 2012.

[89] Tao Tang, Xuejun Yang, and Yisong Lin. Cache miss analysis for gpu programs based on stack distance profile. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 623–634, June 2011.

[90] 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.

[91] Y. Torres, A. Gonzalez-Escribano, and D.R. Llanos. Understanding the impact of cuda tuning techniques for fermi. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 631–639, July 2011.

[92] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A modified approach to data cache management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, MICRO 28, pages 93–103, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[93] Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-Mei W.

Hwu. Languages and compilers for parallel computing. chapter CUDA-Lite: Reducing GPU Programming Complexity, pages 1–15. Springer-Verlag, Berlin, Heidelberg, 2008.

[94] Swapneela Unkule, Christopher Shaltz, and Apan Qasem. Automatic restructuring of gpu kernels for exploiting inter-thread data locality. In *Proceedings of the 21st International Conference on Compiler Construction*, CC'12, pages 21–40, Berlin, Heidelberg, 2012. Springer-Verlag.

[95] 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.

[96] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 199–, Washington, DC, USA, 2002. IEEE Computer Society.

[97] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 430–441, New York, NY, USA, 2011. ACM.

[98] Yuejian Xie and Gabriel H Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 174–183. ACM, 2009.

[99] Yi Yang. Architectural support and compiler optimization for many-core architectures. 2013.

[100] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM.

[101] Chenjie Yu and Peter Petrov. Adaptive multi-threading for dynamic workloads in embedded multiprocessors. In *Proceedings of the 23rd Symposium on Integrated Circuits and System Design*, SBCCI '10, pages 67–72, New York, NY, USA, 2010. ACM.

[102] 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.

[103] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102. ACM, 2009.

Vita

Dong Li received the Bachelor of Science degree in Computer Sciences from the University of Science and Technology of China in July 2001. He then entered the Institute of Computing Technology of the Chinese Academy of Sciences and received a Mater degree in Computer Sciences in July 2004. He was accepted and started graduate studies at the Department of Computer Science at the University of Texas at Austin in August, 2006.

Permanent address: Apt 131, Building No. 72, the 11th District,
Shihezi, Xinjiang, Peoples Republic of China,
832000

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.