Title
Improving Hardware Multithreading in General Purpose Graphics Processing Units

Author
Kim, Hyun Jin

Publication Date
2017

Peer reviewed|Thesis/dissertation

University of California

Los Angeles

Improving Hardware Multithreading in General Purpose Graphics Processing Units

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Hyun Jin Kim

2017

ABSTRACT OF THE DISSERTATION

Improving Hardware Multithreading in
General Purpose Graphics Processing Units

by

Hyun Jin Kim
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2017
Professor Glenn D. Reinman, Chair

General-purpose graphics processing unit (GPGPU) is one of the most popular many-core accelerators that deliver a massive computing power in parallel applications. GPGPUs mainly rely on the hardware multithreading to hide a short pipeline stall and a long memory latency. Thus, the performance of GPGPU can be significantly affected by how GPGPU's hardware multithreading is applied. However, finding the optimal hardware multithreading is a complex problem since there are many aspects to be considered. This work studies the mechanisms for improving the effectiveness of hardware multithreading. First, it studies the various scheduling policies and proposes an adaptive scheduling policy that chooses the best scheduling policy at runtime. In addition, it proposes simple but effective warp throttling mechanism that can increase the cache locality. Furthermore, it proposes a hardware prefetching mechanism to extend the memory latency hiding degree of hardware multithreading. Finally, it shows how a limited scalability of the conventional cache miss handling architecture constrains the degree of hardware multithreading and proposes the highly scalable cache miss handling architecture.

The dissertation of Hyun Jin Kim is approved.

Luminita Vese

D. Stott Parker

Milos Ercegovac

Glenn D. Reinman, Committee Chair

University of California, Los Angeles

2017

Table of Contents

List of Figures

List of Tables

Vita

Education

1998–2005	BA in Computer Science & Engineering, SoongSil University, Seoul, South Korea.
2009–2012	MS in Computer Science, University of California, Los Angeles
2012–Present	Ph.D. Candidate in Computer Science, University of California, Los Angeles

Work Experience

2004–2006	Hardware engineer at Coreriver Semiconductor, Seoul, Korea: RTL Design, Design Validation, Design Testbench, and Low Yield Analysis
2012–Present	Teaching Assistant in Computer Science Department at UCLA: CS33 Computer Organization, CS51A Logic Design of Digital Systems, CS151B Computer System Architecture, and CS180 Algorithms and Complexity

Awards

2003	Gold prize at the Software Competition hosted by SoongSil University
2004	Silver prize at the Software Competition hosted by SoongSil University
2005	Employee of the Year awarded by Coreriver semiconductor

CHAPTER 1

Introduction

General-purpose graphics processing units (GPGPUs) provide tremendous computing power in applications with very high thread-level parallelism (TLP) compared to traditional sequential processors due to GPGPUs' throughput oriented architecture [LNO08, NVI09, KW12, KDK11]. Instead of few complicated cores and large caches like conventional CPUs, most of silicon area is consumed by many simple single-instruction multiple-data (SIMD) cores in GPGPUs to provide high computing throughput. To exploit this high computing power of GPGPUs, programming interface such as CUDA [NVI11b] and OpenCL [Mun13] help programmers develop an application with very high TLP.

Figure 1.1 shows an overview of the GPGPU architecture. A GPGPU has many simple in-order cores called streaming multiprocessor (SM), and each SM consists of 8 to 32 width of SIMD unit and on-chip storage. The latency of SIMD unit varies depending on the types of arithmetic operations and operands data types [WPS10]. Moreover, the SIMD unit is associated with a scoreboard to detect data dependencies among pipeline stages. Each SM has on-chip storage consisting of a register file, a scratchpad memory (shared memory), a private L1 data cache, a private L1 instruction cache, a read-only texture cache, and a constant caches. Threads within thread block can communicate via shared memory. A piece of shared L2 cache is connected to a memory controller. The write-evict and write-back policy is used for L1 and L2 data caches respectively [ROA12]. All the caches have miss status holding registers (MSHR) [Kro98] to support non-block loads. Each piece of L2 cache is connected to SMs via interconnection network. Additionally, GPGPUs provide multiple memory channels to provide high DRAM bandwidth.

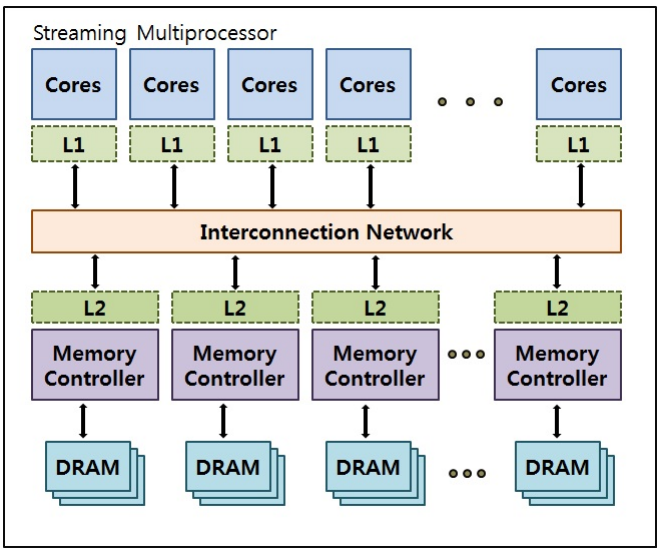

Figure 1.1: Overview of GPGPU Architecture

To efficiently exploit thousands of active threads, GPGPUs use the Single Instruction Multiple Thread (SIMT) [LNO08] execution model that groups a fixed number of threads (typically 32) into a warp (or a wavefront [AMD12]). A warp is a unit of SM's scheduling, and threads in a warp are executed together on GPGPU's SIMD cores. This scalar front end (fetch and decode units) leads the simplification of the pipeline implementation. A warp scheduler decides which warp to execute among multiple ready warps.

In CUDA model, a GPGPU executes a CUDA kernel consisting of a grid of threads. All threads are grouped in multiple thread blocks. A thread block is a unit of thread communication and synchronization. Threads within a thread block can communicate via the shared memory (the scratchpad memory) and perform barrier operations. Each SM can execute multiple thread blocks concurrently, but the maximum number of thread blocks assigned to each SM depends on the size of SM's various resources such as the size of the register file or the size of the shared memory. Multiple thread blocks in a SM are executed independently of one another.

GPGPUs hide pipeline stall cycles and long memory latencies by warp-based hardware multithreading commonly with up to 32˜64 warps. However, GPGPU's performance can be varied depending on warp scheduling policies because the degree of latency hiding depends

on warp scheduling policies.

To reduce long memory latencies, GPGPUs use on-chip storage, such as non-transparent scratchpad memory and cache. The scratchpad memory significantly reduces the memory latency as well as the required DRAM bandwidth if the working set of an application fits the size of the scratchpad memory. Still, the scratchpad memory is ineffective if the working set size of an application is larger than the size of the scratchpad memory. To deal with this problem, GPGPUs have brought a two-level cache since NVIDIA's Fermi [NVI09] architecture. The efficiency of cache, however, can be significantly affected by how warps are scheduled since each warp can compete against other warps in the small size cache. Thus, warp scheduling policies should consider its impact on caches.

Lastly, GPGPU's massively hardware multithreading requires a scalable cache miss handling architecture (MHA) that is also called a lock-up free cache [Kro98]. However, the scalability of conventional MHA is limited in term of an area and a power consumption due to its fully-associative structure. CPUs support only a small size of MHA due to the MSHR's limited scalability. For example, Pentium 4 provides only 8 size of MHA in L1 cache [BBH]. Still, the implementation details of GPGPU's MHA is not officially released.

This chapter introduces the microarchitecture of GPGPUs that can affect the implementation of warp scheduling, and provides the explanation of MHA's mechanism and its limited scalability.

1.1 Microarchitecture of GPGPUs

This section explains three important features that can influence the effectiveness of hardware multithreading: a scoreboard, the ALU pipeline latency, and the physical instruction set.

1.1.1 Scoreboard

A scoreboard keeps track of data dependencies among pipeline stages, and informs which warp can be issued (which warp does not have the data dependency in the pipeline) to the warp scheduler. Then, the warp scheduler can choose the next warp among ready warps. Without a scoreboard, the only way to avoid the data dependency is to fill each pipeline stage with different warp's instructions. Therefore, a warp issued an instruction cannot be scheduled again until the warp's issued instruction is committed without a scoreboard.

Hardware multithreading without the scoreboarding is also called barrel processing. Barrel processing was widely used as a baseline warp scheduler [KJK12, JKC13, JKM13, NSL11, CTY13]. Due to lack of the scorebarding, barrel processing forces the warp scheduler to choose the round robin (RR) scheduling policy. A warp scheduler with the RR policy switches another warp (usually the next warp) in every cycle among ready warps to avoid data dependencies among pipeline stages. The RR policy results in similar execution progress rates of warps. Warps' similar progress rates can improve the cache data locality if a locality exists between warps. It also can increase row buffer hits when multiple warps' memory requests access the same row in DRAM. However, most of the warps confront a long-latency memory instructions almost at the same time in the RR policy [JKC13, NSL11]. This results in a very low degree of long memory latency hiding since only single instruction per warp is usually used to hide the long memory latency.

On the other hand, the warp scheduler can choose various scheduling policies with the scoreboarding since it allows to execute instructions from any ready warps that have no data hazard. Furthermore, it is believed that commercial GPGPUs have a scoreboard [AF].

1.1.2 ALU pipelines

Arithmetic logic units (ALU) in GPGPUs are highly pipelined to increase the computing throughput. A yypical latency of arithmetic instructions is about 22 cycles [NVI11b], and instruction latencies are much larger in complex operations such as arithmetic division or

Operation	**Type**	**Exe. Unit**	**Latency** (clocks)	**Throughput** (ops/clock)
add, sub, max, min	unit, int	SP	24	7.9
mad	uint, int	SP	120	1.4
mul	uint, int	SP	96	1.7
div	uint	-	608	0.28
div	int	-	684	0.23
rem	uint	-	728	0.24
rem	int	-	784	0.20
and, or, xor, shl, shr	uint	SP	24	7.9
add, sub, max, min	float	SP	24	7.9
mad	float	SP	24	7.9
mul	float	SP, SFU	24	11.2
div	float	-	137	1.5
add, sub, max, min	double	DPU	48	1.0
mad	double	DPU	48	1.0
mul	double	DPU	48	1.0
div	double	-	1366	0.063

Table 1.1: Latency and Throughput of Arithmetic and Logic Operations [WPS10]

#Loads	1	2	3	4	5	#Kernels
PTX	127	15	1	3	0	16
PTXPlus	164	43	7	11	3	47

Table 1.2: The distribution of single and consecutive loads from 86 kernels' assembly codes that have consecutive loads.

transcendental operations [WPS10].

Table 1.1 shows the ALU's latency and throughput of various arithmetic and logic operations in GT200 [WPS10]. GT200 is an outdated GPGPU (NVIDIA's first GPGPU generation). Thus, less latency and higher throughput are expected in recent GPGPUs, especially division operations which are processed with multiple operations. Still, most of arithmetic and logic instructions require more than half of all the warps (32 warps in our GPGPU model) to fill ALU pipelines. Thus, the warp scheduling should be properly designed to fill this long ALU pipeline.

1.1.3 Compiler Optimization

In the CUDA model, the NVIDIA's GPGPU compiler *nvcc* compiles the GPGPU code section into the Parallel Thread eXecution (PTX) instruction set that is a virtual instruction set. Then the assembler *ptxas* assembles PTX into the native instructions. When the native instruction is generated, the compiler optimization is applied. Instead of PTX, we used a PTXPlus [AF] that is a extensive version of PTX. The effect of the compiler optimization in native instructions is applied to the PTXPlus. Regard to the memory latency hiding, a noticeable difference between PTX and the native instruction is the number of consecutive loads. The PTXPlus code has larger number of consecutive loads than the PTX code as shown in Table 1.2.

From 83 GPGPU kernels, 37 kernels have consecutive loads in the PTXPlus, but only 16 kernels have consecutive loads in the PTX as shown in Table 1.2. The total number of

consecutive loads with the PTXPlus is 64, but it is just 19 with the PTX. Most consecutive loads have no data dependency between them. Thus, memory requests from consecutive loads can be issued together. Issuing consecutive loads together can overlap each load's memory latency. However, it is difficult for barrel processing to exploit the overlapping memory latency of consecutive loads because the barrel processing switches to the next warp in each load instruction.

1.2 Cache Miss Handling Architecture

GPGPU's hardware multithreading requires an efficient cache miss handling architecture (MHA) that is also called a lock-up free cache [Kro98]. A lock-up free cache supports non-blocking load misses that enable GPGPUs (or CPUs) to continue executing instructions upon a load miss until any subsequent instruction has a data dependency on the load miss. the lockup-free cache keeps the information of a load miss in the miss status holding register (MSHR) [Kro98] to support non-blocking loads.

When SM executes a memory instruction, fewer wide memory requests can be issued instead of generating individual memory request per a thread within a warp through the memory access *coalescing* (also called *intra-warp memory coalescing*). The number of memory requests per a warp varies from a single memory request to 32 memory requests depending on the address distance between threads' memory requests and the memory block size of each request. If each thread in a warp accesses diverse range of addresses upon a single memory instruction, then the single memory instruction can result in issuing multiple memory requests that is called the memory divergence.

GPGPUs also support *inter-warp memory coalescing* [BYF09] to reduce the memory traffic via MSHRs. When multiple warps access the same address and miss a cache, then only a single memory request is issued by the inter-warp memory coalescing instead of issuing multiple memory requests with the same address. To support the inter-warp memory coalescing, each MSHR entry maintains multiple fields. The total number of in-flight memory

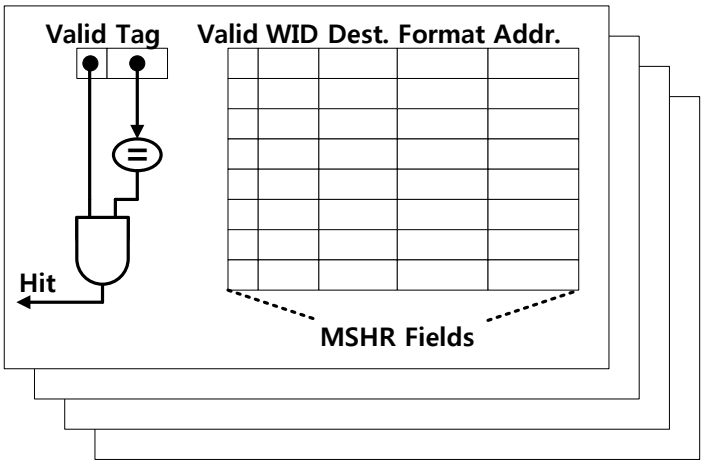

Figure 1.2: Conventional MSHR Structure: 4 MSHR entries (8 fileds/entry)

requests is limited by the number of ready MSHR entries.

Figure 1.2 shows the conventional MSHR structure that supports non-blocking loads. MSHRs consists of many MSHR entries. Each MSHR entry has a valid bit, a tag, and multiple MSHR fields. It also has its own tag comparator that is used in a fully-associative search upon a load miss. A MSHR field consists of a valid bit, a warp ID, destination that is the target register number, a format of load instruction, and the address in the cache block. The GPGPU's MSHR structure is almost identical to MSHR used in CPUs except for the WID tag in the MSHR field. We assume the WID tag in the MSHR field from the fact that GPGPUs access the registers for a warp by a warp ID and register ID[NSL11]. We also assume that the destination includes the register ID as a target register number and 32-bit valid bit for each thread's register. Since all the 32 registers are accessed together in a warp, 10 bit address (5 bit from warp ID and 5-bit from register ID) is used for 32K size register file. Thus, the size of an MSHR field is 1 (valid) + 5 (warp ID) + 5 (register ID) + 32 (valid for each register) + 6 (format) + 7 (address in a block) = 55-bit in our assumption. We use the explicily addressed MSHR [FJ94].

Load misses are categorized into primary misses and secondary misses in non-blocking loads. A primary miss is the first cache miss to a cache block. When a primary miss occurs,

one of ready MSHR entries is consumed. The primary miss sets the value bit and tag in the MSHR entry and fill the first MSHR field. A secondary cache miss is any subsequent cache miss on the cache block that already has a primary miss. The secondary misses can be merged in the MSHR entry that handled the corresponding primary miss. A MSHR entry has multiple fields to support non-blocking loads for many secondary misses. If there is no available MSHR entry or MSHR field on a primary miss or a secondary miss, respectively, the cache is locked-up and it rejects further load misses. When the cache is locked-up, the processor has to stall upon further load miss.

A memory request issued by a non-blocking load consumes one MSHR entry. Thus, the number of MSHR entries decides the maximum number of memory requests that can be issued. GPGPUs need many MSHR entries to fully exploit the hardware multithreading since each warp can issue multiple memory requests, but MSHRs are not scalable in term of area and power due to the MSHRs' fully-associative structure; all the MSHR entries are searched associatively upon a load miss [TCT06]. CPUs support only few number of MSHRs due to the MSHR's limited scalability. For example, Pentium 4 provides only 8 MSHRs in L1 cache [BBH]. C. Nugteren [NBC14] claimed that L1 data cache has 64 MSHR entries in NVIDIA's GTX470. GPGPU-Sim [BYF09] uses 32 MSHRs in both L1 and L2 data caches though the number of MSHRs in GPGPUs is not officially released.

CHAPTER 2

Locality-Aware Warp Scheduling Policy

This chapter studies the effects of various warp scheduling policies in general-purpose graphics processing units (GPGPUs). The performances of warp scheduling policies can vary depending on the characteristics of a workload's data locality. This study proposes Locality-Aware Warp Scheduling (LAWS) to improve the performance of GPGPUs by adaptively changing the warp scheduling policy depending on a workload's data locality characteristics. An evaluation shows that LAWS results in a geometric mean of 18% performance improvement over GTO in 16 scheduling sensitive benchmarks. Overall, LAWS improves the performance by 7% over GTO scheduling policy in 42 benchmarks including scheduling policy insensitive benchmarks.

2.1 Limitation of Barrel Processing

Barrel processing forces the warp scheduler to switch to the next warp in every cycle due to the lack of a scoreboard. Thus, the RR warp scheduling policy naturally fits in the barrel processing. The warps' execution progress rates in a SM are similar in the RR policy, and these similar progress rates can be helpful to catch the data locality between warps in the cache and the DRAM (row buffer hits). However, the RR policy provides a limited latency hiding degree because of the warps' similar progress rates.

To increase the limited latency hiding degree in the barrel processing, many two-level based barrel processing (2LEV) were proposed in the literature [JKC13, JKM13, NSL11, CTY13]. The 2LEV forms a few groups of warps that each group has the same number of

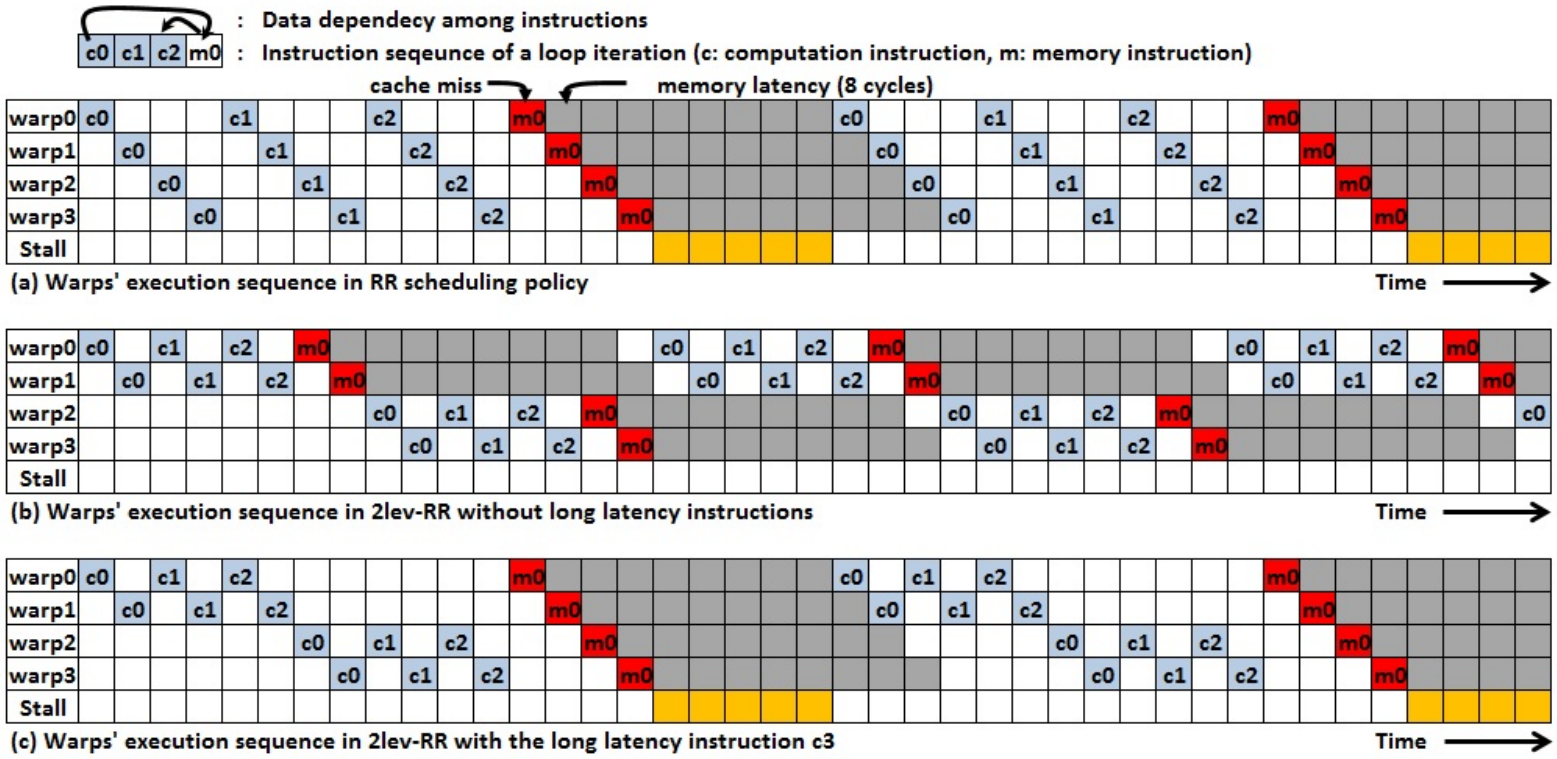

Figure 2.1: Examples of warps' execution sequence with RR and 2LEV polices.

warps. A warp-group in 2LEV can be a fixed number of warps [NSL11] or single thread block [JKC13, JKM13, CTY13]. Instead of selecting the next warp from all ready warps like the normal barrel processing, the 2LEV gives higher priority to warps within the same warp-group until the scheduler has to switch to the next warp-group due to out of ready warps. The scheduler switches to the next warp-group when there is no ready warps in the current warp-group. The scheduling policy is still the RR in both the warp scheduling and the warp-group scheduling. The 2LEV can provide much higher degree of latency hiding than the barrel processing by exploiting most of other warp-groups' instructions to hide the current warp-group's latency. However, 2LEV did not consider the GPGPU's highly-pipelined ALU that takes many warps to fill as discussed in section 1.1.2. The highly-pipelined ALU's long latency requires frequent warp-group switches in 2LEV; it diminishes the effectiveness of 2LEV.

Figure 2.1 shows the 2LEV's improved latency hiding degree over the normal barrel processing and limited latency hiding degree by a long latency instruction. For simplicity, it assumes 4 warps (2 warps per a warp-group), the 2-stage pipeline, and 8-cycle memory latency. As shown in Figure 2.1(a) and (b), the barrel processing hides only 3 cycles from 8-cycle memory latency while the 2LEV hides all the latency by the other warp-group's instructions. However, 2LEV's latency hiding degree can be limited by a long latency instruction as shown in Figure 2.1 (c). We assume that c2 is 4-cycle latency instruction in Figure 2.1 (c). The warp scheduler has to switch to the next warp-group upon the long latency instruction. This c2's long latency reduces the 2LEV's latency hiding cycles from 8 to 3 cycles, same as the barrel processing.

A warp scheduler with the scoreboard can choose various scheduling policies unlike the barrel processing. Consecutive loads are also an important factor in the scheduling policy decision since more overlapping long-memory latency is required to hide consecutive loads' long-latency. Figure 2.2 shows the effects of the scoreboard and consecutive loads. For simplicity, it assumes 4 warps, the 4-stage pipeline, and 8-cycle memory latency. The RR with a scoreboard can provide a higher degree of latency hiding than the barrel processing

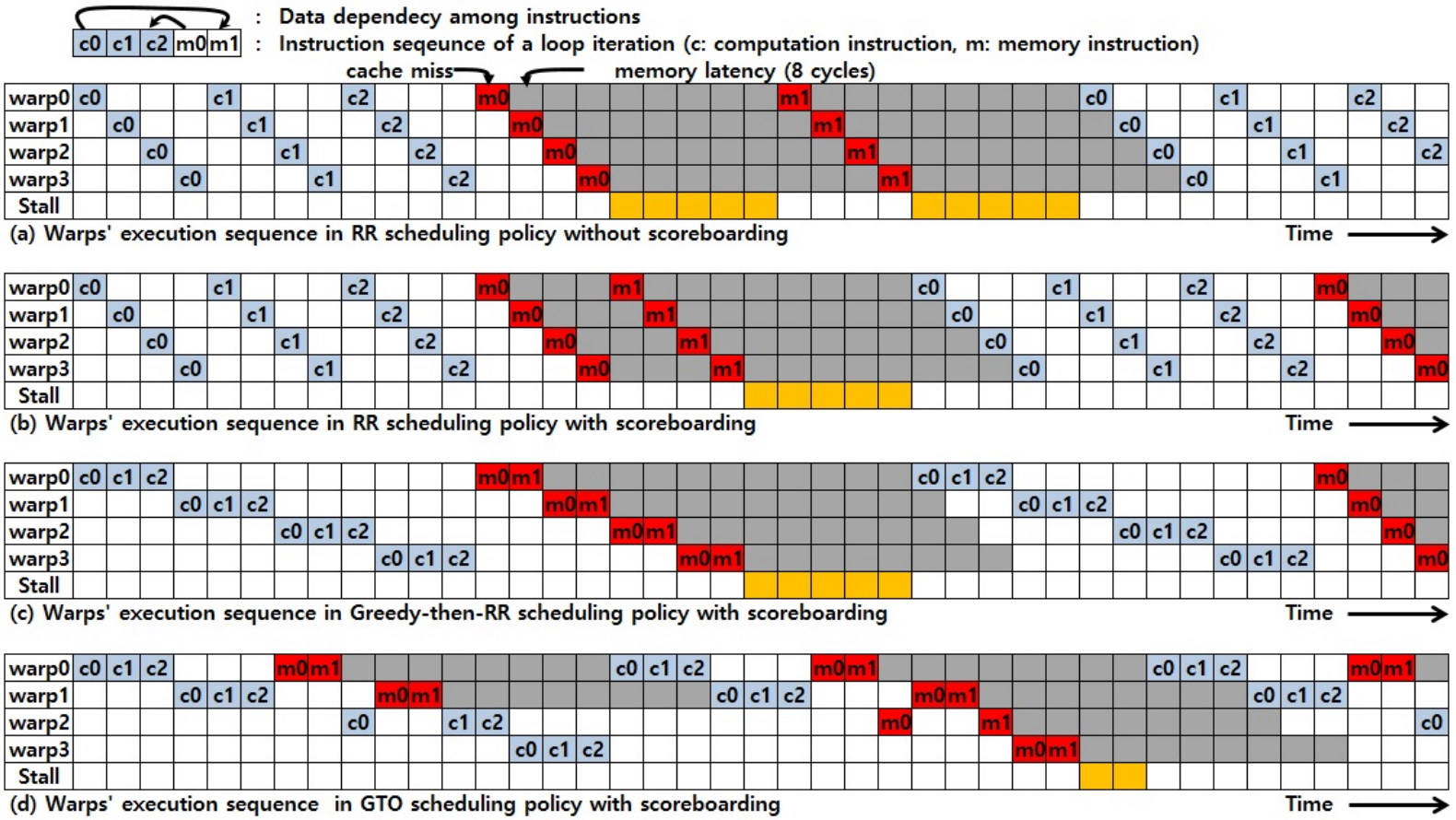

Figure 2.2: Examples of warps' execution sequence in RR and GTO w/o a scoreboard.

by exploiting more instruction-level parallelism as shown in Figure 2.2 (a) and (b). The RR with a scoreboard overlaps m0 and m1 instructions' memory latency that results in the half stall cycles of the RR without a scoreboard. The warp scheduler can choose various scheduling policies as shown in Figure 2.2 (c) and (d): a greedy-then-RR (GTRR) policy and a simple fixed priority policy (SFP), respectively. The GTRR tries to keep scheduling the same warp until it detects a data hazard, then it switches to the next warp like the RR. In the SFP, warps are prioritized based on the warp ID (the warp0 has the highest priority). SFP tries to scheduler a higher priority warp's instructions until it detects the data hazard between them.

As shown in Figure 2.1 and 2.2, simple barrel processing and modified scheduling policies based on the barrel processing [JKC13, JKM13, NSL11, CTY13] have limitations to provide a high long-latency hiding degree for GPGPUs. To devise better warp scheduling policies, the effects of a scoreboard, long-latency ALU, and consecutive loads are considered to devise a better warp scheduling policy.

2.2 Motivation

The effectiveness of the warp scheduler can be significantly affected by underlying microarchitecture (a scoreboard and an ALU pipeline) and an compiler optimization (consecutive loads). Still, a warp scheduling policy is an important factor to decide the performance of the warp scheduler including the three factors.

The warp scheduler with fair policies, such as the RR policy, leads to almost even execution rates of warps. Thus, fair policies can efficiently capture both cache locality among warps (*inter-warp locality*) and row-buffer locality in DRAM [NSL11, JKC13]. However, it results in a low degree of memory latency hiding. On the other hand, a warp scheduler can pick an unfair scheduling policy. A warp scheduling with an unfair policy can try to keep executing a higher priority warp than a lower priority warp until it has to switch. This unfairness results in varying the progress rates of warps that can provide more instructions

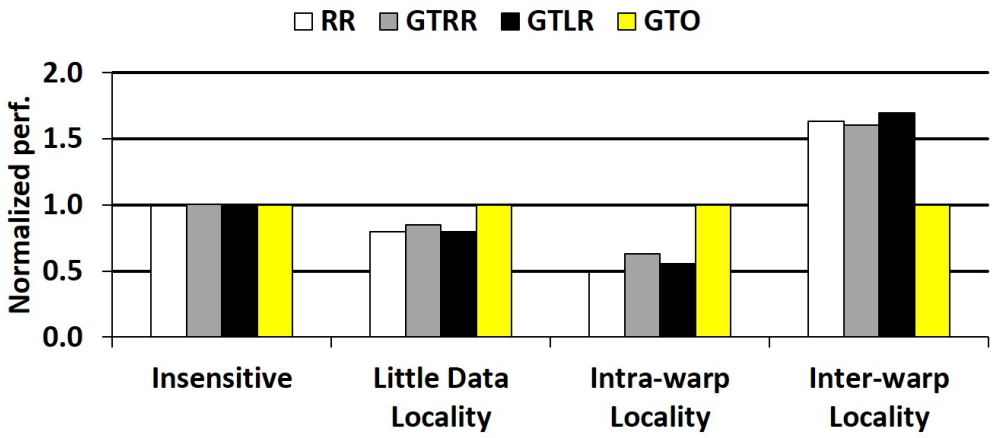

Figure 2.3: Performance of various warp scheduling policies normalized to GTO policy in four different benchmark groups.

to hide long memory latency than fair scheduling policies. It can also capture more cache locality within a warp (*intra-warp locality*) than fair policies [ROA12] in the L1 data cache.

The data locality in the L1 data cache can be classified in two types of localities in GPGPUs: intra-warp locality and inter-warp locality [ROA12, ROA13]. When a warp hits a cache block hits that is filled by the same warp, the cache block is said to have intra-warp locality. If the cache block is hit by another warp, then it is considered to have inter-warp locality.

Figure 2.3 shows the performance with three fair scheduling policies (RR, GTRR, and GTLR) over an unfair scheduling policy, GTO [ROA12]. Section 2.2 explains these scheduling policies in detail. Benchmarks are classified into four difference groups in Figure 2.3. If a benchmark's performance is not significantly affected by scheduling policies, then it is classified in the insensitive group. Otherwise, a benchmark is categorized based on its L1 data cache locality characteristics. A benchmark with high cache miss rate is classified in the little data locality group. The rest of the benchmarks are classified in the intra-warp or inter-warp locality group based on the majority of data localities.

GTO shows slightly higher performance than fair policies in the little data locality group since GTO's unfairness can provide a higher degree of memory latency hiding than fair

policies. Moreover, GTO shows significantly higher performance than fair policies in the intra-warp locality group. However, fair scheduling policies show better performance than GTO in the inter-warp locality group. This result motivates us to devise an adaptive warp scheduling that effectively captures both intra- and inter-warp data locality .

2.3 Warp Scheduling Policies

This study makes use of the following scheduling policies: 3 fair scheduling policies (RR, GTRR, GTLR), a unfair policy (GTO), two-level scheduling policies (2LEV), and the proposed locality-aware warp scheduling (LAWS).

RR Simple round-robin warp scheduling policy. The priorities of warps are ordered in round-robin fashion.

GTRR A greedy then round-robin policy. The scheduler keeps executing a warp until it detects a data hazard. If a data hazard is detected in the scoreboard, then the scheduler switches to another ready warp.

GTLR A greedy then round-robin on a memory instruction. This is similar to GTRR except that if a warp issues an load instruction, the scheduler switches to the next ready warp. GTLR is a hybrid of RR and GTRR. It uses the greedy policy, but the scheduler switches to next ready warp upon a memory instruction to capture the inter-warp data locality.

GTO A greedy then oldest scheduling policy [ROA12]. The scheduler keeps executing a warp until it detects a data hazard then it switches to the oldest (the highest priority) ready warp. The age of warp (priority) is determined with the assignment time of the thread block and the warp ID number. Within a thread block, a warp with smaller warp ID has a higher scheduling priority. Between thread blocks, all warps in a thread block assigned earlier have higher priority than all warps in a thread block assigned later.

2LEV A two-level warp scheduling [NSL11]. The scheduler divides the entire warps into few fetch groups, and puts higher priorities on warps in a selected fetch group than warps in other fetch groups. Thus, the scheduler keeps scheduling the warps in the fetch group, and then switches to the next ready fetch group if all warps in the fetch group are stalled. In our experiment, we use 8 warps as the fetch group size, same as the original study [NSL11]. All four scheduling policies described above are applied to schedule warps in a fetch group. We examined two policies (RR and GTO) for the fetch group selection policy.

LAWS Our locality-aware adaptive warp scheduling that changes its scheduling policy based on the locality score system.

2.4 Warp Scheduling Policy on Different Data Locality

The degree of data locality captured in the L1 data cache can be significantly affected by warp scheduling policies. Figure 2.4 compares the effectiveness of the fair warp scheduling policy (RR) and the unfair warp scheduling policy (GTO) on different types of data localities. We assume the followings for the simplicity in Figure 2.4: 4-stage pipeline, 3-cycle memory latency on a cache miss, two cache blocks, and two MSHR entries. The two cache blocks are initially filled by warp0's m0 and warp1's m0.

2.4.1 Intra-warp Locality

Cache blocks with the intra-warp locality can be easily evicted by interference of other warps. This intra-warp locality loss is usually larger in fair scheduling policies than in unfair scheduling policies because fair policies switch between warps more frequently than unfair policies as shown in Figure 2.4 (a) and (b). In Figure 2.4 (a) and (b), we assume that there are intra-warp locality between m0 and m1 in all warps over loop iterations. In the RR policy, warp0 and warp1 hit the cache when they execute m0 instructions, then the intra-

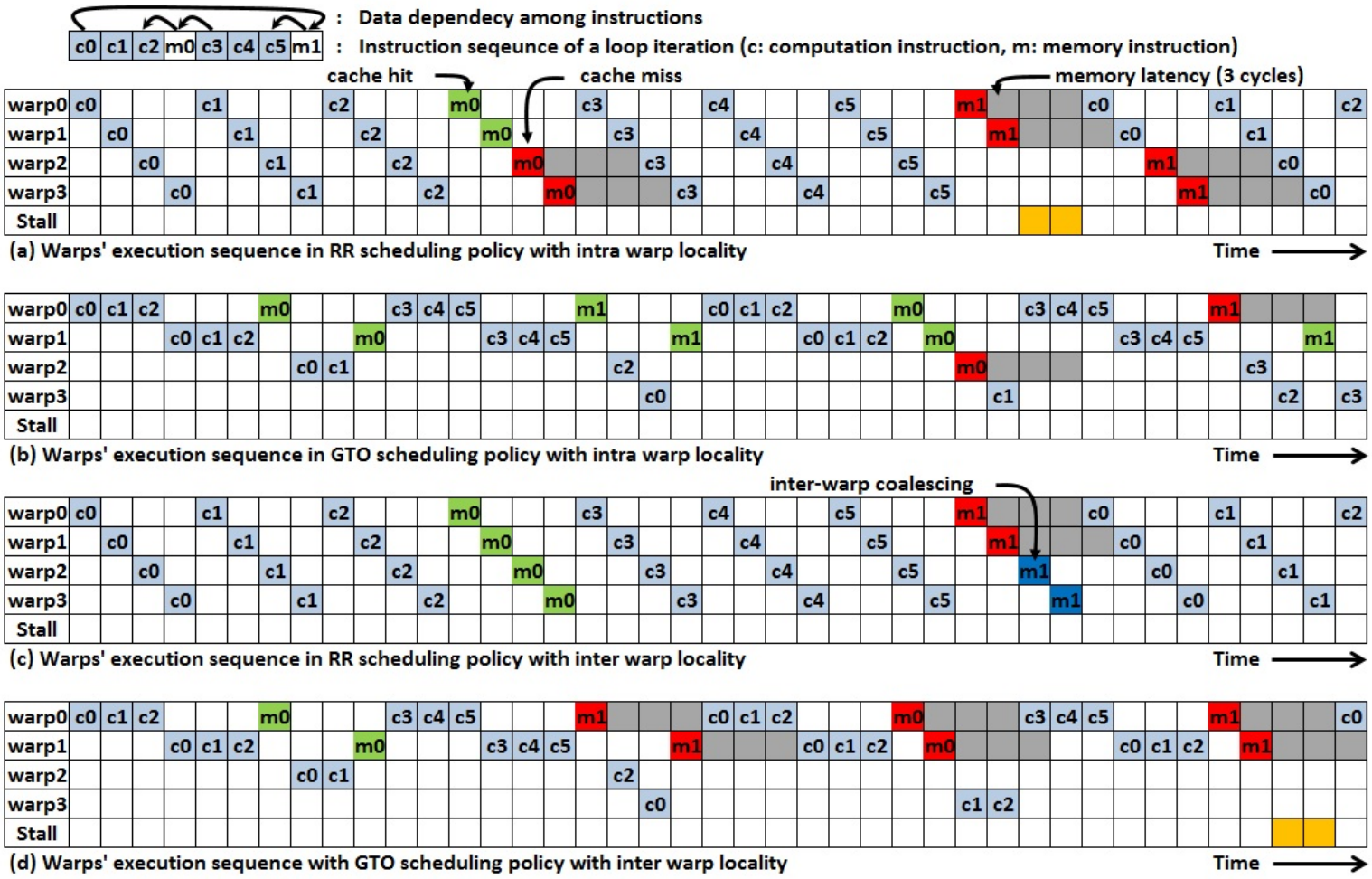

Figure 2.4: Examples of warps' execution sequence with RR and GTO polices on intra-/inter-warp data locality.

warp localities are lost because the cache blocks are replaced by warp2's m0 and warp3's m0. Thus, warp0's m1 and warp1's m1 cause cache misses as shown in Figure 2.4 (a). However, the GTO policy captures the intra-warp locality better than the RR policy since the initial cache blocks are kept longer due to the higher priority of warp0 and warp1 over warp2 and warp3 as shown in Figure 2.4 (b). Furthermore, if the scheduler reduces the number of warps to 2 (warp0 and warp1), then the two cache misses in Figure 2.4 (b) would not occur. This explains how throttling can keep more intra-warp locality than simple unfair policies.

2.4.2 Inter-warp Locality

Fair scheduling policies can capture the inter-warp locality more than unfair scheduling policies since the progress rates of warps are similar in fair policies. In Figure 2.4 (c) and (d), we assume that there is an inter-warp locality in m0 and m1 instructions among even numbered warps (warp0, warp2) and odd numbered warps (warp1, warp3) respectively. All warps hit the L1D cache in RR policy when they execute m0 instructions due to the inter-warp locality. Then, warps miss the cache when they execute m1 instructions. However, only two memory requests are issued for all four m1 instructions because memory requests of warp2's m1 and warp3's m1 are merged in memory requests of warp0's m1 and warp1's m1 respectively via MSHR due to the inter-warp locality. On the other hand, m0's inter-warp localities are lost in the GTO policy because the initial cache blocks are evicted by warp0's m1 and warp1's m1 before warp2's m0 and warp3's m0 are executed. Moreover, there is no inter-warp memory coalescing in the GTO policy due to the variance of the warps' progress rates.

2.5 Locality-Aware Scheduling

This section describes our locality-aware warp scheduling (LAWS). The warp scheduler has the locality score system (LLS) as shown in Figure 2.5. The LLS delivers the type of the data locality that is abundant. Based on the data locality characteristics measured in the

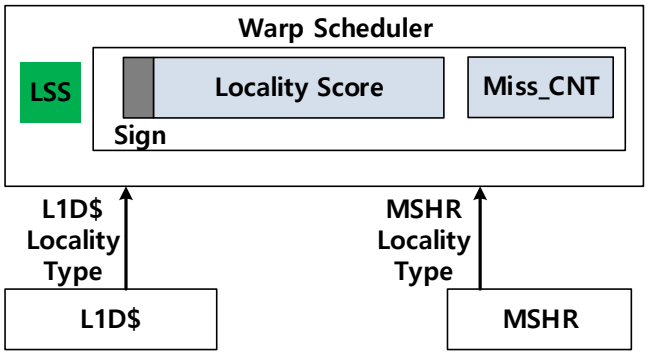

Figure 2.5: LAWS Mechanism

LSS, the warp scheduler chooses its scheduling policy.

2.5.1 Locality Scoring System

LAWS chooses either a fair or an unfair scheduling policy based on the locality score in the LSS. The LSS has a 16-bit signed integer as the locality score and 5-bit Miss_CNT register as shown in Figure 2.5. When a kernel is launched, the locality score is initialized to 0. The locality score increases on a primary miss [Kro98] or a hit by the intra-warp locality detected in L1 data cache or MSHR. On the other hand, the score decreases on a hit by the inter-warp locality detected in the L1 data cache and MSHRs. Thus, the locality score can represent which locality type is dominant in a workload at runtime. A positive locality score indicates that the workload has a little data locality or a high intra-warp data locality. Otherwise, a negative locality score means that the workload has a high inter-warp data locality. The warp scheduler checks the sign bit of the locality score, then it chooses a fair policy if the sign bit is 1, otherwise it chooses an unfair scheduling policy. LAWS uses the GTO for its unfair scheduling policy. We choose the GTLR policy for LAWS's fair policy since the GTLR shows the best performance in applications with a high inter-warp data locality as discussed in section 2.6.1. A primary miss or a cache/MSHR hit by the intra-warp locality increases the locality score by 1. However, we put more weight on the inter-warp locality value to compensate the locality score increment by primary misses. The LSS counts the number of

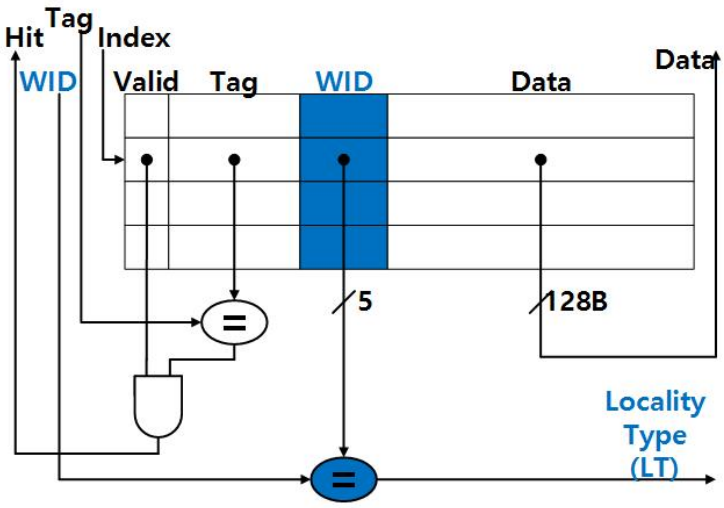

Figure 2.6: Data Locality Detection Mechanism in L1 Data Cache

primary misses in the Miss_CNT. A cache/MSHR hit by the inter-warp locality decreases the locality score by Miss_CNT value+1 and resets the Miss_CNT to 0. We choose 31 for the maximum Miss_CNT value (5-bit size) based on its sensitivity experiments in section 2.7.3.2. To detect the data locality type of a cache hit, we add the warp ID tag (WID) in each L1 data cache block as shown in Figure 2.6. The WID represents the warp ID that brought in the cache block due to a primary miss. Thus, the WID is updated by a warp's primary miss. A cache hit and its WID tag hit indicate the hit by the intra-warp locality. It sets the locality type signal (LT) to 1. A cache hit and its WID tag miss indicate the hit by the inter-warp locality. It sets the LT to 0. The 2-bit locality information upon a cache access (Hit, LT) is delivered from the L1 data cache to the LSS.

A secondary cache miss [Kro98] is any subsequent cache miss on a cache block that already has a primary miss. The secondary misses can be merged in the MSHR entry (an MSHR hit) that handled the corresponding primary miss. We also consider the locality types of MSHR hits for the locality score since they could be cache hits if they are scheduled at different time. To detect an MSHR hit's type of the data locality, we compare the warp ID (WID) of the first MSHR field in an MSHR entry to the warp ID of an MSHR access (a secondary miss) as shown in Figure 2.7 since the first field in an MSHR entry is filled by a

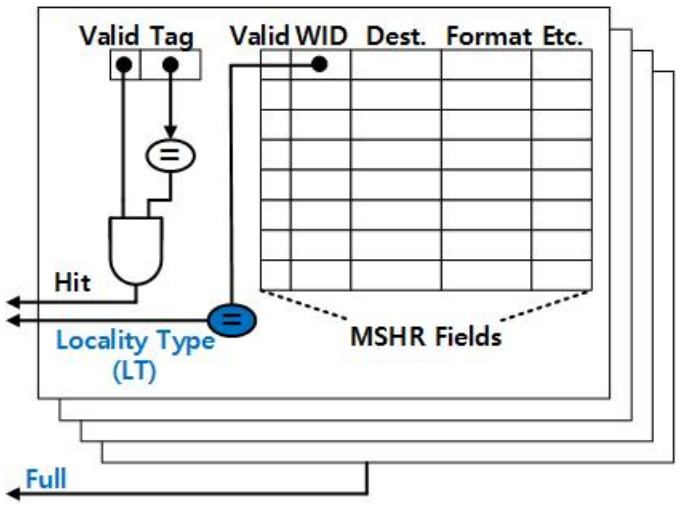

Figure 2.7: MSHR Modification

primary miss. Similar to the L1 data cache, the 2-bit (Hit, LT) is delivered to the LSS. In MSHRs, we assume the WID tag in the MSHR field from that GPGPUs access registers by a warp ID and a register number since the warp ID is also used in indexing the register file in GPGPUs [NSL11].

2.6 Experimental Methodology

We extend GPGPU-Sim 3.2.1 [BYF09] to support various warp scheduling policies and LAWS. The configuration details are shown in Table 2.1. We study 42 CUDA applications from six different benchmark suites: CUDA SDK [NVI11a], GPGPU-Sim benchmark [BYF09], Rodinia [CBM09a], Parboil [SRS12], SHOC [DMM10], and MAR [HFL08]. These applications are categorized in four groups based on scheduling policy sensitivity and the characteristics of the data locality in the L1 data cache: type-0(insensitive), type-I(little data locality), type-II(intra-warp data locality), and type-III (inter-warp data locality).

If a benchmark's maximum performance difference between various scheduling policies and the GTO is less than 10%, it is considered type-0. Twenty six benchmarks are classified type-0, including the following: CUDA SDK(convolutionSeparable, fastWalshTransform, matrixMul, reduction, scalarProd, scan, sortingNetwork, transpose), GPGPU-Sim bench-

Parameter	Value
# Compute Units	30
SIMD unit width	32
Warp Size	32
# Threads / Core	max 1024
# Registers / Core	32768
Shared Memory / Core	32KB (16 banks)
Constant Cache / Core	8KB (2-way, 64B line, LRU)
Texture Cache / Core	16KB (8-way, 128B line)
L1 Data Cache	32 KB (8-way, 128B line, LRU)
MSHR in L1D cache	32 entries (8 fields / entry)
Shared L2 cache	128KB / MC (16-way, 128B line, LRU),
MSHR in L2D cache	32 entries (8 fields / entry)
Core Clock	1400 Mhz
Interconnect Clock	1400 Mhz
Memory Clock	924 Mhz
Interconnection	crossbar, 32B channel width
# Memory Channels	8
Memory Controller	FR-FCFS [RDK00]
DRAM request queue size	32
DRAM	16 DRAM banks / MC
GDDR5 Timing	tCL=12, tRP=12, tRC=40, tRAS=28, tRCD=12, tRRD=6
Memory Channel BW	8 (Bytes/Cycle)

Table 2.1: Simulator Configuration

Type	Name	Abbr.	#Ker.	CTAs	T/CTA
I	AES Cryptography [BYF09]	AES	1	256	256
	Coulombic Potential [BYF09]	CP	1	256	128
	Histogram [NVI11a]	HIST	4	-	-
	Hotspot [CBM09a]	HS	1	1849	256
	Sort [DMM10]	SORT	5	-	-
II	Similarity Score [DMM10]	SS	19	-	-
	StoreCPU [BYF09]	STO	1	1536	127
	Breadth First Search [CBM09a]	BFS	2	256	512
	Kmeans* [ROA12]	KMEAN	1	121	256
	Sparse Matrix Vector Multiplication [DMM10]	SPMV-s (scalar version)	1	256	128
	Inverted Index [HFL08]	II	17	-	-
III	SRAD_ver1 [CBM09a]	SRAD1	7	-	-
	CFD solver [CBM09a]	CFD	4	-	-
	Monte-Carlo [NVI11a]	MONT	2	-	-
	3D Laplace Solver [BYF09]	LPS	1	2048	128
	stream-Cluster [CBM09a]	SC	1	128	512

Table 2.2: GPGPU Benchmarks Description:Kmeans* = Modified Kmeans from Rodinia [CBM09a] benchmark suite used in CCWS [ROA12]

.

mark(LIB, NN), rodinia (backprop, gaussian, heartwall, lud, nw, srad_v2), Parboil(cutcp, lbm, mri-q, sad, sgemm, stencil), and SHOC(FFT, QTC, S3D, SPMV-vector). The remaining 16 benchmarks are categorized based on its L1 data cache locality characteristics as shown in Table 2.2. Table 2.2 shows the number of kernels (#Ker.), the number of thread blocks (#CTAs), and the number of threads per a thread block (T/CTA) of a benchmark. If the primary cache miss rate of an application is more than 95%, the application is classified into the little data locality group. If the primary cache miss rate is less or equal to 95%, the application is classified into the intra- or inter-warp locality group based on the majority of data localities.

2.7 Experimental Results

2.7.1 Scheduling Policy Performance

Various scheduling policies' performance in the scheduling policy sensitive benchmarks are shown in Figure 2.8. The GTO shows higher performance than fair scheduling policies (RR, GTRR, and GTLR) in type-I and type-II as shown in Figure 2.8(a) and (b). On the other hand, fair policies show higher performance than the GTO in type-III as shown in Figure 2.8(c). Among three fair policies, we choose the GTLR for LAWS's unfair policy since it shows the highest performance in type-III. LAWS reaches the performance of the best scheduling policies across all types. By adaptively changing the scheduling policy between the GTO and the GTLR, LAWS closely reaches GTO's performance in type-I and type-II and GTLR's performance in type-III. The overall performance of LAWS outperforms the GTO by 7% (18% excluding type-0) as shown Figure 2.9.

Performances of the 2LEV with various policy combinations are shown in Figure 2.10. 2R and 2G uses RR and GTO for the fetch group selection policy, respectively. Overall, all 8 different combinations in the 2LEV reach about 80% of LAWS's performance. No combination reaches LAWS's performance in both type-II and type-III. This indicates that simply mixing fair and unfair scheduling policy does not effectively capture both intra- and

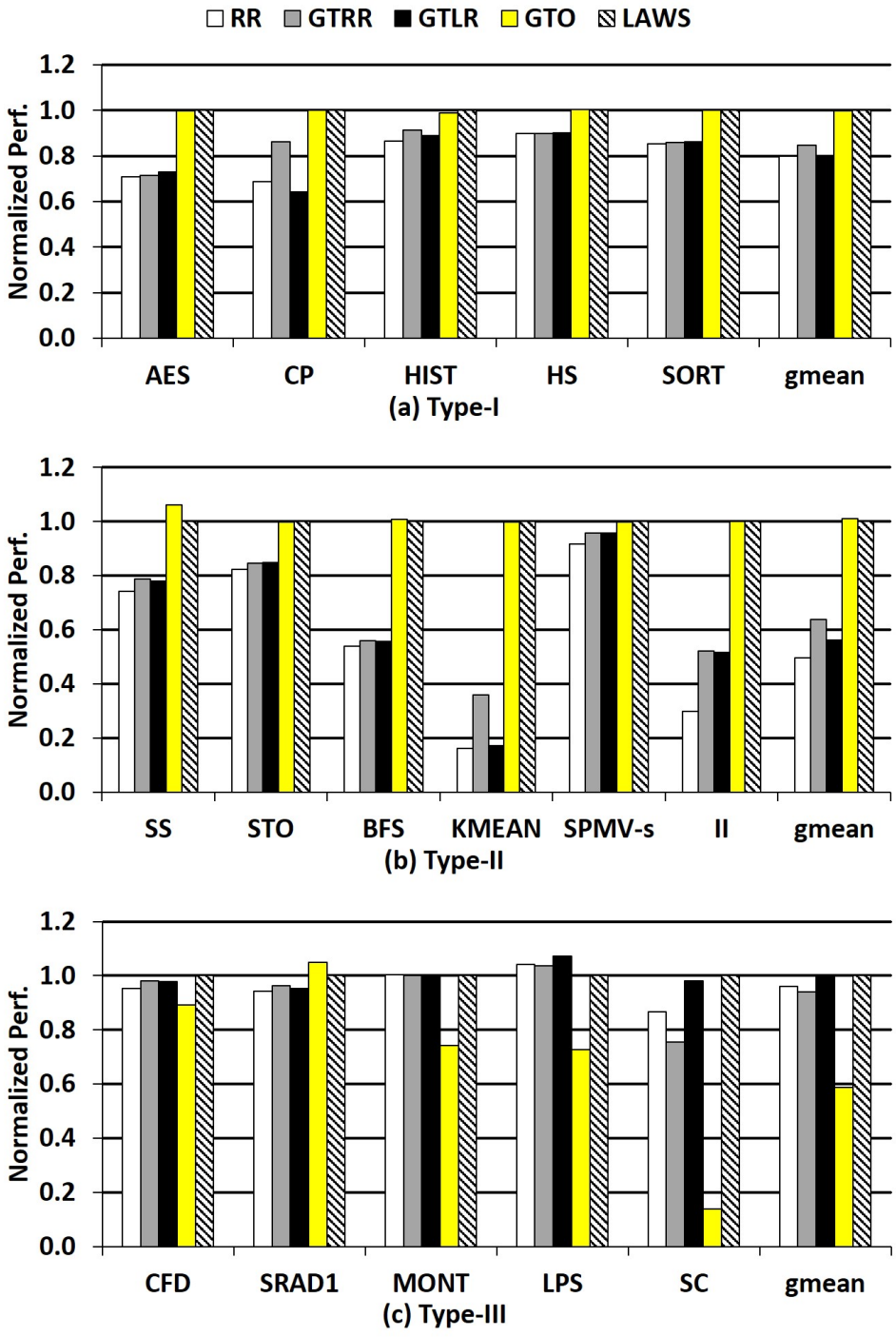

Figure 2.8: Performance of 16 scheduling policy sensitive benchmarks on various scheduling policies and throttling mechanisms normalized to LAWS

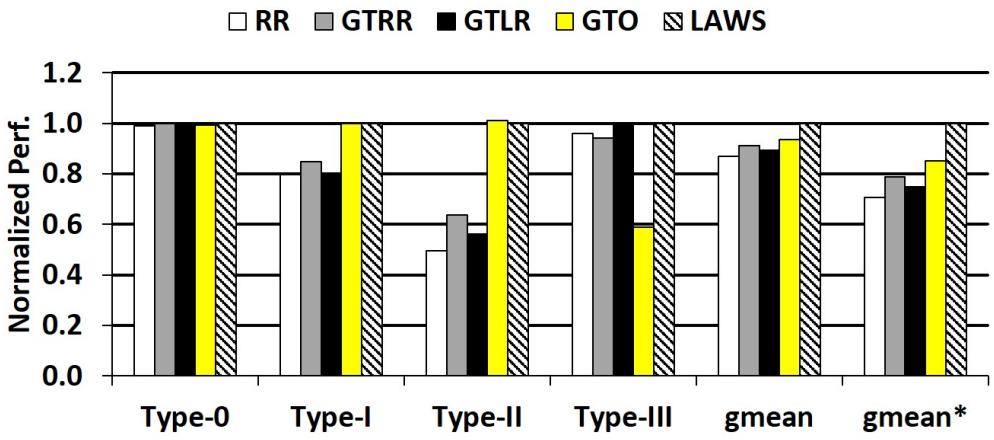

Figure 2.9: Overall performance of various scheduling policies and throttling mechanisms normalized to LAWS (gmean*: geometric mean except for type-0)

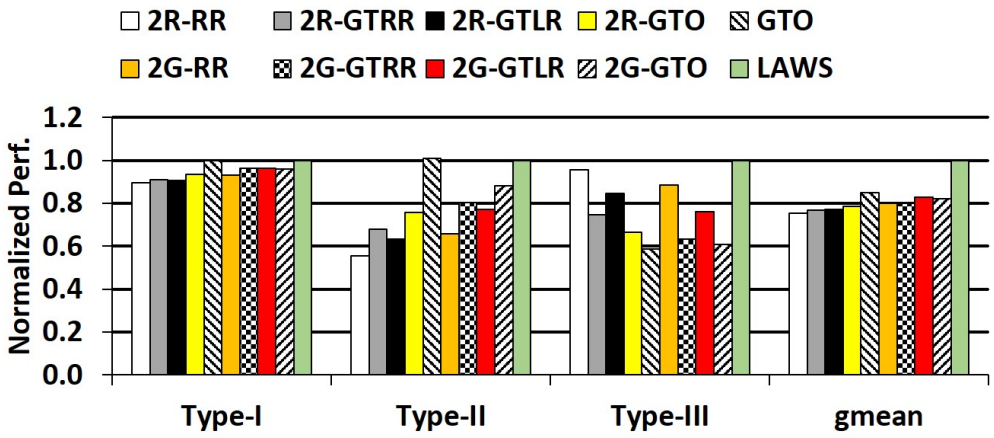

Figure 2.10: Overall Performance of 2LEV with various policies normalized to LAWS

inter-warp locality since each policy can offset the other policy's advantage.

2.7.2 Data Locality Analysis

Figure 2.11 shows the L1 data cache access classification with various scheduling policies: primary miss (MISS), hit with intra-warp locality (INTRA), hit with inter-warp locality (INTER), secondary miss (or hit on MSHR) with intra-warp locality (INTRA-M), and secondary miss with inter-warp locality (INTER-M). Overall, the GTO captures more intra-warp locality than the GTLR, and the GTLR captures more inter-warp locality than the GTO.

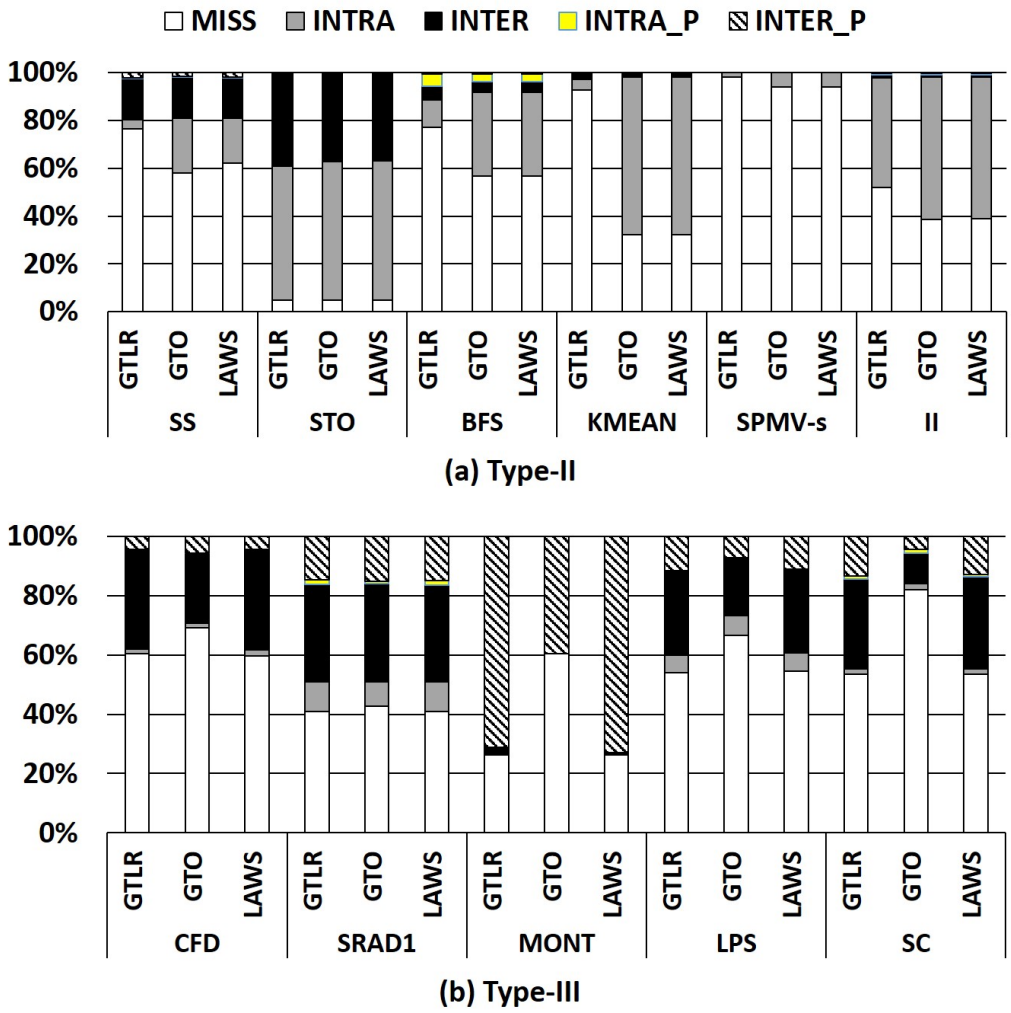

Figure 2.11: Classification of L1 data cache accesses

Scheduling policies have little impact on the L1 cache locality in STO and SRAD1 since the L1 data cache size (32K) is enough to hold their memory footprint.

2.7.3 LAWS Sensitivity

2.7.3.1 Input Data Sensitivity

LAWS provides a significant performance improvement over GTO in three type-III benchmarks: MONT, LPS, and SC as shown in Figure 2.8(c). Still, the performance improvement can be varied depending on the input data size or input data configuration since they can

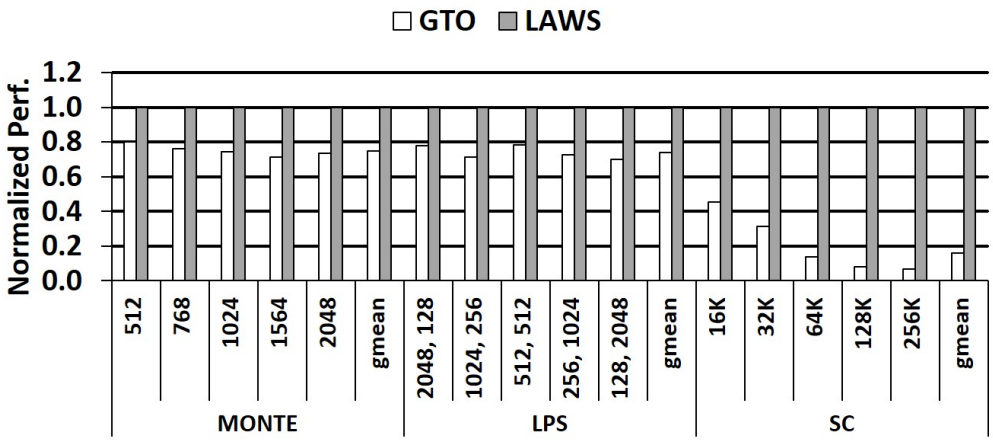

Figure 2.12: Performance results of five input configurations in MONT, LPS, and SC normalized to LAWS.

Config	MONT	LPS	SC
A	512	2048,128	16K
B	768	1024,256	32K
C	1024	512,512	64K
D	1564	256,1024	128K
E	2048	128,2048	256K

Table 2.3: Five input configuration of MONT, LPS, and SC: MONT(optionData size), LPS(nx, ny), SC(Number of data points)

affect the data structure of a workload. Figure 2.12 shows LAWS's performance improvement over GTO in five different input configurations of these three benchmarks. We vary the option data size of MONT, the combination of nx and ny value of LPS, and the number of data points (data size) of SC in this experiment. We use the middle input configuration from the five configurations for LAWS and LAWS-TH experiments. These input configurations are 1024, (512, 512), and 64K in MONT, LPS, and SC, respectively.

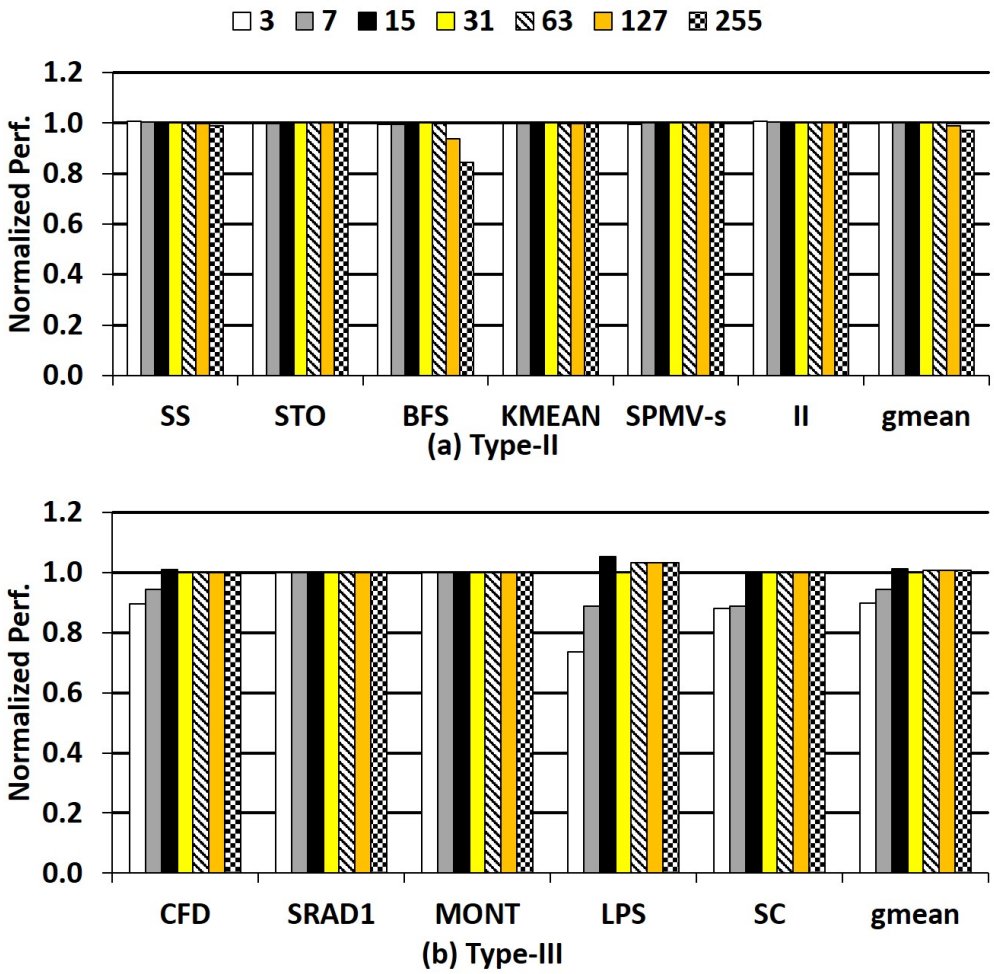

Figure 2.13: LAWS performance with varying the maximum Miss_CNT value normalized to 31

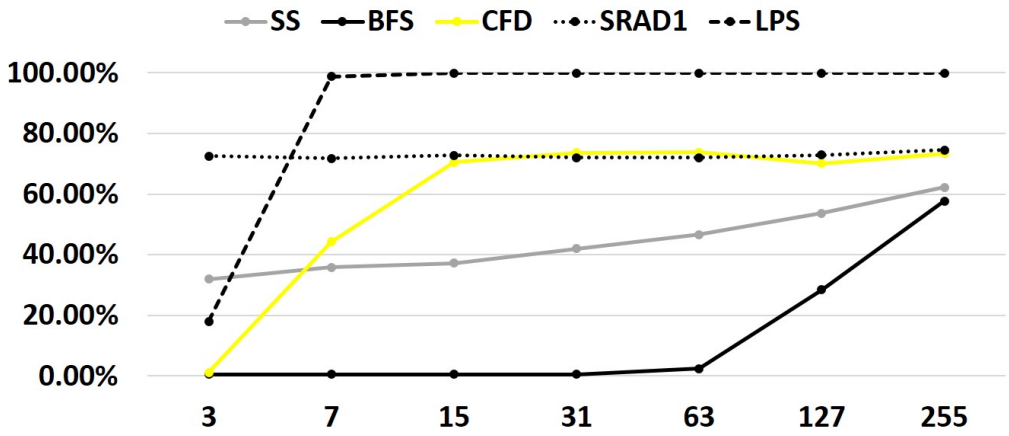

Figure 2.14: Ratio of the execution time with GTLR policy on varying the maximum Miss_CNT value

2.7.3.2 Miss_CNT Size Sensitivity

LAWS puts a more weight on the inter-warp locality using Miss_CNT. However, if the weight is too high, then it can hurt the performance of the type-II since LAWS might switch to GTLR unnecessarily. On the other hand, if the weight is too low, it can hurt the performance of type-III since LAWS takes more time to switch to GTLR. Figure 2.13 shows LAWS's performance with varying the maximum Miss_CNT values normalized to the performance with a maximum value of 31 in type-II and type-III applications. The GTRL execution time ratios with varying maximum Miss_CNT values are also shown in Figure 2.14. In Figure 2.14, the ratio of STO, KMEAN, SPMV-s, II, MONT, and SC are not shown: the ratio is less than 1% in STO, KMEAN, SPMV-s, and II, and the ratio is large than 99% in MONT and SC across all Miss_CNT sizes.

Type-II benchmarks' performances are not significantly affected by varying the maximum Miss_CNT except for BFS. BFS's performance starts to decrease with 127 since LAWS's the ratio of GTRL policy in BFS starts to increases at 127 as shown in Figure 2.14. Still, 127 is too large weight for inter-warp locality. Up to 63, the performance of type-II are not affected. On the other hand, SS stays its maximum performance while SS's GTLR ratio is increased as Miss_CNT increases from 32% to 62%. The SS's performance is barely affected

by the GTRL's varying ratios because SS's inter-warp locality is not affected by scheduling policies (L1 data cache size is enough to keep most SS's inter-warp locality) as shown in Figure 2.11(a).

In type-III benchmarks, SRAD1 and MONT retain their maximum performances across all the Miss_CNT values since their high GTRL ratios are barely affected by increasing the Miss_CNT value. On the other hand, performances of CFD, LPS, and STM are increased as the Miss_CNT value increases up to 16, then they maintain the maximum performance after 16. These increasing performances come from the increased GTLR ratios as shown in Figure 2.14 except for SC. SC's GTRL ratio reaches more than 99% across all the Miss_CNT values, but it results in 90% of the maximum performance at 4 and 8 Miss_CNT values. These 10% performance drops of SC at the small Miss_CNT values comes from frequent switching between GTO and GTLR that increases SC's entire execution time. In sum, LAWS maintains type-II and type-III benchmarks' maximum performances in the wide ranges of Miss_CNT values from 15 to 63. We use 31 for LAWS.

2.7.4 Implementation Complexity

LAWS consists of two main components: a warp scheduler with the locality scoring system (LSS) and the locality detection unit in the L1 data cache and MSHR. Most of the implementation cost comes from the 5-bit WID tag in the L1 data cache. The 5-bit WID per each cache line requires 160 bytes of storage in each SM's 32K L1 data cache. We believe the implementation cost of the rest of the components are negligible since they require only a few bits of storage and few logic gates.

2.8 Related Work

2.8.1 Warp Scheduling Policy

Lakshminarayana et al. [LK10] explore various instruction fetch policies based on fair scheduling policies. However, their fair-based policies' latency hiding degrees are limited compared to unfair scheduling policies.

Narasiman et al. [NSL11] propose the two-level scheduling policy to increase the degree of memory latency hiding in a barrel processing. The two-level policy can provide higher performance than RR policy in barrel processing. However, simple unfair policies can achieve such high degree of latency hiding in our GPGPU model as discussed in section 2.7.1.

A memory region-based warp scheduling policy is proposed by Chen et al. [CTY13]. Their scheduling policy shifts warps' priority based on the analysis of memory regions done by a compiler. However, their work focuses on memory latency hiding, and does not consider the effect of scheduling policies on the data locality in the cache. Gebhart et al. [GJT11] propose a two-level warp scheduling mechanism to reduce the energy consumption for GPGPUs.

2.8.2 Thread Block Scheduling

Kayiran et al. [KJK12] introduce a dynamic CTA (thread block) scheduling to improve the performance of GPGPUs by reducing the contention on the memory subsystem in the memory intensive applications. In contrast, our work improves the performance by capturing more data localities in the L1D cache. Jog et al. [JKC13] propose thread block based scheduling policies based on a two-level scheduling policy to exploit data locality within each thread block in the L1 data cache and bank-level parallelism in the DRAM.

Lee et al. [LSM14] present an alternative thread block scheduling policies with GTO as a warp scheduling policy. Their thread block scheduling policy dynamically controls the number of thread blocks per core and assign consecutive thread blocks to the same core to improve performance. However, they only use the GTO policy for warp scheduling that

leads to poor performance in applications with high inter-warp data locality.

We remain thread block scheduling with LAWS in the future work. However, we believe that LAWS can be improved further with more efficient thread block scheduling since warp scheduling and thread block scheduling are independent of one another.

2.9 Conclusion

This work evaluates the performance of various warp scheduling policies in different types of GPGPU applications. It classifies GPGPU applications based on the data locality characteristics, and it analyzes the performance of warp scheduling policies based on fairness. It demonstrates how unfair and fair policies effectively capture intra-warp locality and inter-warp locality respectively.

Based on this observation, we propose Locality-Aware Warp Scheduling to exploit the benefits of both fair and unfair scheduling policies. LAWS is a novel technique that dynamically changes its scheduling policy based on the locality characteristics at runtime. In this work, we implement locality-warp adaptive scheduling with the very small cost.

CHAPTER 3

Warp Throttling

Thousands of threads share the small size L1 data cache (L1D) of each core (streaming multiprocessor) in GPGPUs. The L1D's data locality can be easily lost by the sever resource contention on the L1D. To reduce such data locality loss in the L1D, we propose a novel throttling mechanism that dynamically decreases the number of warps depending on the warp-based data locality degree in the L1D. It exploits miss status holding registers (MSHR) to measure the degree of intra-warp locality in the L1D and the degree of memory divergence. Based on these measurements, it controls the number of warps for scheduling. Our evaluation shows that LAWS with throttling results in a geometric mean of 58% performance improvement over GTO scheduling policy in 16 scheduling sensitive benchmarks. Overall, LAWS with throttling improves the performance by 19% over GTO scheduling policy respectively in 42 benchmarks including scheduling policy insensitive benchmarks.

3.1 Motivation

A high intra-warp locality can be easily lost by cache misses from other warps. Therefore, the number of memory requests by cache misses should be minimized to maintain a high intra-warp locality [ROA12, ROA13]. Furthermore, the intra-warp locality lost in the L1 data cache becomes worse if a high memory divergence exists that can cause multiple memory requests to be issued from single warp's memory instruction [ROA13].

Figure 3.1 shows the effectiveness of throttling in different types of applications. B-SWL, which schedules statically optimal number of warps, outperforms GTO significantly in the

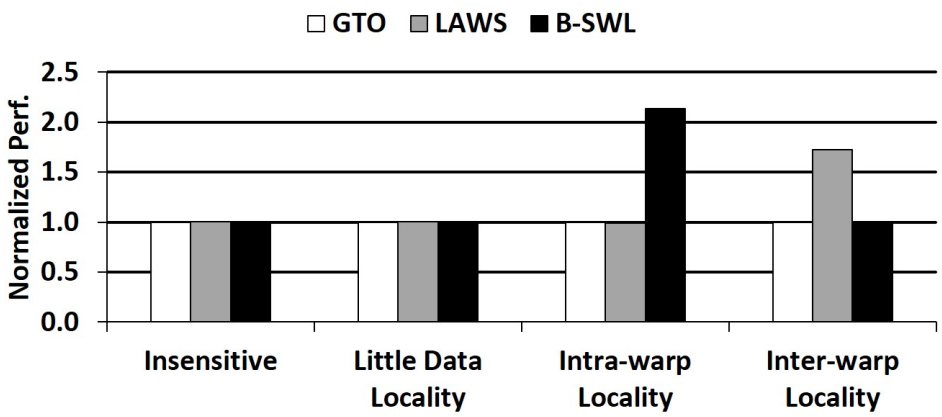

Figure 3.1: Performance of various warp scheduling policies normalized to GTO policy in four different benchmark groups.

intra-warp locality group by reducing cache interference among warps. However, the B-SWL does not improve the performance of applications with inter-warp locality unlike LAWS. This result leads to devise a throttling mechanism coupled with the LAWS that minimizes the cache locality loss while effectively capturing both intra- and inter-warp data locality. This study proposes a simple adaptive throttling mechanism based on the measurement of the intra-warp locality degree and the cache interference degree by exploiting MSHRs. By coupling with the LAWS, the throttling mechanism does not only capture the inter-warp locality effectively but also reduces the implementation cost significantly.

3.2 Various Warp Throttling

This study makes use of the following warp throttling methods: best static warp limiting (Best-SWL), cache-conscious wavefront scheduling (CCWS) [ROA12], LAWS-TH, and LAWS-NDM.

Best-SWL Static Warp Limiting [ROA12, ROA13] uses a fixed number (32 to 1) as the maximum number of warps for scheduling. Best-SWL runs the full range of warp limiting numbers then chooses the optimal number that provides the highest performance. The GTO used for the scheduling policy in Best-SWL same as the original

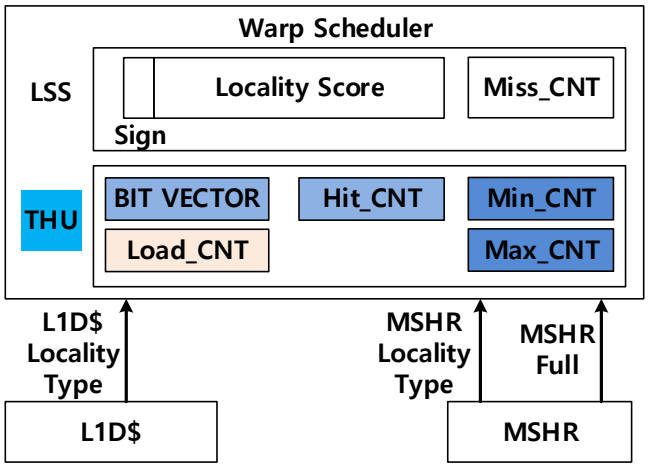

Figure 3.2: Overview of Throttling Unit on LAWS

work [ROA12, ROA13].

CCWS Cache-Conscious Wavefront Scheduling (CCWS) uses the GTO for its warp scheduling policy. However, CCWS reduces the intra-warp locality loss caused by other warps' cache interference in the L1 data cache. It throttles the number of warps for scheduling when it detects the eviction of useful cache blocks. It adds a victim tag array in the L1 data cache to detect the intra-warp locality loss at runtime [ROA12].

LAWS-TH LAWS with our throttling mechanism

LAWS-NMD LAWS-TH without throttling by memory divergence

3.3 Throttling Unit

The throttling unit (THU) shown in Figure 3.2 measures the degree of the intra-warp locality and the memory divergence. The THU is added on the LAWS, and it reuses the locality detection units in the L1D and MSHRs. When the MSHRs become full, the THU decides the number of warps for scheduling. It exploits fully consumed MSHRs for the throttling decision timing for two reasons. First, it provides the minimum intra-warp locality degree

at runtime. Second, the THU can easily measure the memory divergence degree at runtime. Once throttling is enabled, the THU throttles the number of warps at a minimum of two warps and maintains the throttling for 1K cycles at least. Then, the THU performs the throttling decision when the MSHRs become full again. The LAWS-TH is the LAWS coupled with the THU. In the LAWS-TH, the throttling is not enabled when it is running with a fair policy (GTRL) since the throttling does not help improve the inter-warp locality degree.

3.3.1 Throttling by Intra-Warp Locality

The THU measures a warp-based intra-warp locality. It has a 32-bit vector to show whether a warp has the intra-warp locality or not. If a warp has a hit in the L1 data cache by the intra-warp locality, the corresponding bit in the bit vector is set to 1. The bit is reset to 0, when the corresponding warp misses the cache or finishes its execution. When MSHRs become full, the maximum number of memory requests (primary cache misses) are issued. Thus, the bit vector can indicate the minimum intra-warp locality upon the fully consumed MSHRs. The THU keeps track of the number of 1's in the bit vector in the Hit_CNT, and it also maintains Hit_CNT's minimum and maximum values in the Min_CNT and Max_CNT, respectively. When all MSHR entries are consumed, THU checks Hit_CNT, Min_CNT and Max_CNT. Then, the THU enables throttling if Hit_CNT >0. The calculation of the number of warps for scheduling is:

$$\#\mathbf{Warp} = \mathbf{Hit\_CNT{+}1\ OR}\ \lceil \mathbf{(Min\_CNT + Max\_CNT)/2} \rceil$$

We observed that an average of Min_CNT and Max_CNT is close to the optimal warp number for the warp throttling. However, it takes few throttling periods to acquire the correct Min_CNT and Max_CNT values. Especially, if warps' memory accesses interfere with each warp's intra-warp locality, then the correct Max_CNT value is available when a proper throttling is enabled. To search the correct Max_CNT value, the THU initially uses Hit_CNT+1 as the number of warps for scheduling until the Max_CNT value is stabilized. We add one warp to Hit_CNT to find the maximum Hit_CNT value of the workload. When the

throttling is enabled, the Hit_CNT is usually increased if all warps' intra-warp locality does not fit the L1D cache. Thus, Hit_CNT+1 eventually passes the optimal number of warps, then starts to consume MSHR entries by cache misses. During this process, if the Hit_CNT becomes larger than the Max_CNT value, the Max_CNT is updated. If the Max_CNT is not updated during the throttling period, the THU realizes that the throttling enters into the stabilized phase. We consider Min_CNT and Max_CNT stabilized if either value is not updated in 8 previous throttling periods as discussed in section 3.5.3.2. Once Min_CNT and Max_CNT are stabilized, the THU uses the ceiling average value for the number of warps.

3.3.2 Throttling by Memory Divergence

A memory divergence degree can be measured in the memory coalescing unit by counting the number of memory accesses to the L1 data cache. However, it can hurt the intra-warp locality only if most memory accesses miss the cache. We devise a simple but novel mechanism to detect a high memory divergence that can hurt the intra-warp locality degree by exploiting the MSHR. Each cache miss consumes an MSHR entry (a primary miss) or an MSHR field (a secondary miss). Thus, if many MSHR entries are consumed by a few number of warps, this indicates that many cache blocks were evicted by the high memory divergence. Based on this observation, the THU keeps track of the number of warps consuming MSHR entries at the Load_CNT as shown in Figure 3.2. When all MSHRs are consumed, the THU is notified. The THU, then, decides the throttling degree based on the following:

#Warp = Load_CNT if Load_CNT <Divergence factor

When MSHRs become full, the THU checks the Load_CNT value for the throttling. If no intra-warp locality is detected (Hit_CNT = 0), the THU sets the number of warps for the scheduling to the Load_CNT if the Load_CNT is less than the memory divergence factor. We use 8 for the memory divergence factor since it indicates a high memory divergence (each warp issues more than 4 memory requests to satisfy the condition). During the throttling by Load_CNT, the Hit_CNT can be increased if the workload also has high intra-warp locality.

Parameter	Value
# Compute Units	30
SIMD unit width	32
Warp Size	32
# Threads / Core	max 1024
# Registers / Core	32768
Shared Memory / Core	32KB (16 banks)
Constant Cache / Core	8KB (2-way, 64B line, LRU)
Texture Cache / Core	16KB (8-way, 128B line)
L1 Data Cache	32 KB (8-way, 128B line, LRU)
MSHR in L1D cache	32 entries (8 fields / entry)
Shared L2 cache	128KB / MC (16-way, 128B line, LRU),
MSHR in L2D cache	32 entries (8 fields / entry)
Core Clock	1400 Mhz
Interconnect Clock	1400 Mhz
Memory Clock	924 Mhz
Interconnection	crossbar, 32B channel width
# Memory Channels	8
Memory Controller	FR-FCFS [RDK00]
DRAM request queue size	32
DRAM	16 DRAM banks / MC
GDDR5 Timing	tCL=12, tRP=12, tRC=40, tRAS=28, tRCD=12, tRRD=6
Memory Channel BW	8 (Bytes/Cycle)

Table 3.1: Simulator Configuration

Type	Name	Abbr.	#Ker.	CTAs	T/CTA
I	AES Cryptography [BYF09]	AES	1	256	256
	Coulombic Potential [BYF09]	CP	1	256	128
	Histogram [NVI11a]	HIST	4	-	-
	Hotspot [CBM09a]	HS	1	1849	256
	Sort [DMM10]	SORT	5	-	-
II	Similarity Score [DMM10]	SS	19	-	-
	StoreCPU [BYF09]	STO	1	1536	127
	Breadth First Search [CBM09a]	BFS	2	256	512
	Kmeans* [ROA12]	KMEAN	1	121	256
	Sparse Matrix Vector Multiplication [DMM10]	SPMV-s (scalar version)	1	256	128
	Inverted Index [HFL08]	II	17	-	-
III	SRAD_ver1 [CBM09a]	SRAD1	7	-	-
	CFD solver [CBM09a]	CFD	4	-	-
	Monte-Carlo [NVI11a]	MONT	2	-	-
	3D Laplace Solver [BYF09]	LPS	1	2048	128
	stream-Cluster [CBM09a]	SC	1	128	512

Table 3.2: GPGPU Benchmarks Description:Kmeans* = Modified Kmeans from Rodinia [CBM09a] benchmark suite used in CCWS [ROA12]

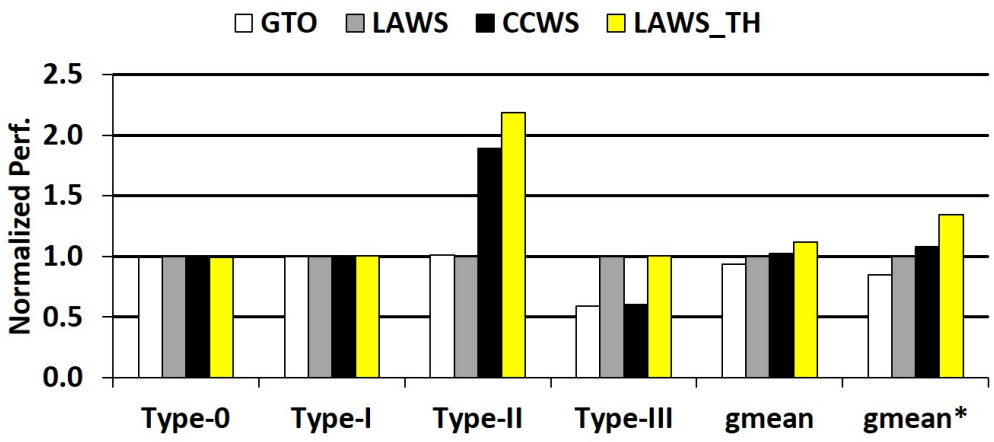

Figure 3.3: Overall performance of various throttling mechanisms normalized to LAWS

If Hit_CNT+1 >Load_CNT, then the THU switches to Hit_CNT+1 for the warp number for throttling instead of Load_CNT.

3.4 Experimental Methodology

We extend GPGPU-Sim 3.2.1 [BYF09] to support B-SWL, CCWS, LAWS, LAWS-TH, and LAWS-NDM. The configuration details are shown in Table 3.1. We study 42 CUDA applications from six different benchmark suites: CUDA SDK [NVI11a], GPGPU-Sim benchmark [BYF09], Rodinia [CBM09a], Parboil [SRS12], SHOC [DMM10], and MAR [HFL08]. These applications are categorized in four groups based on scheduling policy sensitivity and the characteristics of the data locality in the L1 data cache: type-0(insensitive), type-I(little data locality), type-II(intra-warp data locality), and type-III (inter-warp data locality).

If a benchmark's maximum performance difference between various scheduling policies and the GTO is less than 10%, it is considered type-0. Twenty six benchmarks are classified type-0, including the following: CUDA SDK(convolutionSeparable, fastWalshTransform, matrixMul, reduction, scalarProd, scan, sortingNetwork, transpose), GPGPU-Sim benchmark(LIB, NN), rodinia (backprop, gaussian, heartwall, lud, nw, srad_v2), Parboil(cutcp, lbm, mri-q, sad, sgemm, stencil), and SHOC(FFT, QTC, S3D, SPMV-vector). The remaining 16 benchmarks are categorized based on its L1 data cache locality characteristics as

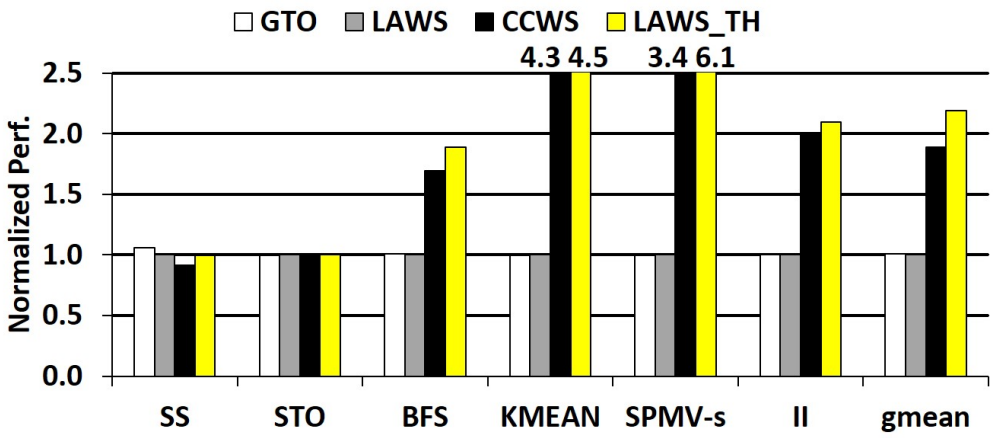

Figure 3.4: Perfromance of various throttling mechanisms in Type-II benchmarks normalized to LAWS

shown in Table 3.2. Table 3.2 shows the number of kernels (#Ker.), the number of thread blocks (#CTAs), and the number of threads per a thread block (T/CTA) of a benchmark. If the primary cache miss rate of an application is more than 95%, even with throttling, the application is classified into the little data locality group. If the primary cache miss rate is less or equal to 95%, the application is classified into the intra- or inter-warp locality group based on the majority of data localities.

3.5 Experimental Results

3.5.1 Throttling Performance

CCWS and LAWS-TH result in the geometric mean of 8% and 35% overall performance improvement over LAWS in the 16 scheduling sensitive benchmarks as shown in Figure 3.3. Applications in type-0 and type-I are not affected by the throttling since CCWS and LAWS are barely enabled. In type-II applications, CCWS and LAWS-TH significantly increase performance by 1.9x and 2.2x times over LAWS, respectively. In type-III applications, LAWS-TH's performance is almost the same as LAWS. However, CCWS shows only 60% of LAWS's performance in type-III applications since CCWS does not consider the inter-warp

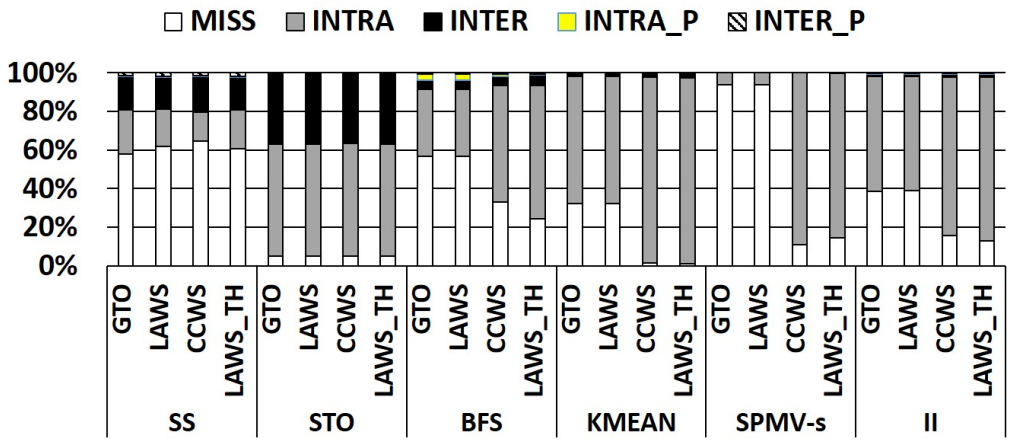

Figure 3.5: Classification of L1 data cache accesses

locality. Section 3.5.3.1 analyzes the throttling performance in different L1 data cache sizes.

3.5.2 Data Locality Analysis

Figure 3.5 shows the L1 data cache access classification with various throttling mechanisms: primary miss (MISS), hit with intra-warp locality (INTRA), hit with inter-warp locality (INTER), secondary miss (or hit on MSHR) with intra-warp locality (INTRA-M), and secondary miss with inter-warp locality (INTER-M). Throttling (CCWS and LAWS-TH) significantly increases the cache hit rate in BFS, KMEAN, SPMV-s, and II over GTO and LAWS as shown in Figure 3.5. Without throttling (GTO and LAWS), SPMV-s results in a very high miss rate (94%) due to the high memory divergence. Interestingly, CCWS results in a lower miss rate (11%) than LAWS-TH (15%) in SPMV-s, but CCWS results in only 55% of the LAWS-TH performance as shown in Figure 3.4. CCWS's lower performance with a lower miss rate demonstrates CCWS's excessive throttling upon high memory divergence as discussed in section 3.5.3.1.

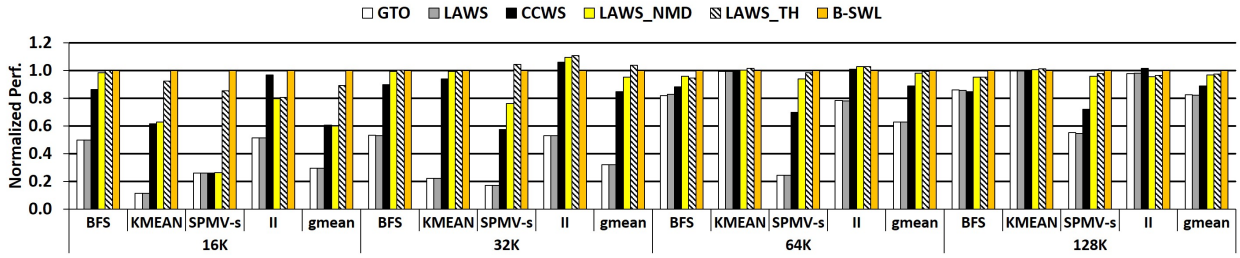

Figure 3.6: Performance of various throttling normalized to B-SWL with varying L1 data cache size.

Benchmarks	16K	32K	64K	128K
BFS	3	5	7	8
KMEAN	3	6	6*	6*
SPMV-s	2	3	5	8
II	3	4	6	8*

Table 3.3: Optimal warp number used in B-SWL (*: no significant difference between B-SWL and no throttling

3.5.3 LAWS-TH Sensitivity

3.5.3.1 Sensitivity to L1 data cache size

Throttling performances in BFS, KMEAN, SPMV-s, and II with varying the L1 data cache size are shown in Figure 3.6. To show the effect of throttling on a high memory divergence, Figure 3.6 includes LAWS-NMD, which is LAWS-TH excluding the throttling by high memory divergence. Optimal warp numbers used in B-SWL are shown in Table 3.3. LAWS-TH closely reaches B-SWL's performance across all the cache sizes. It provides 89%, 104%, 99%, and 98% of B-SWL performance in 16K, 32K, 64K, and 128K, respectively. However, B-SWL does not always present the best performance. Especially, LAWS-TH and CCWS show 6% and 11% higher performances over B-SWL in II with 32K, and LAWS-TH outperforms B-SWL by 5% in SPMV-s with 32K. Overall, LAWS-TH also shows the highest performance among CCWS, LAWS-NMD, and LAWS-TH in all the cache sizes. Overall, the benefit of throttling decreases as the cache size increases.

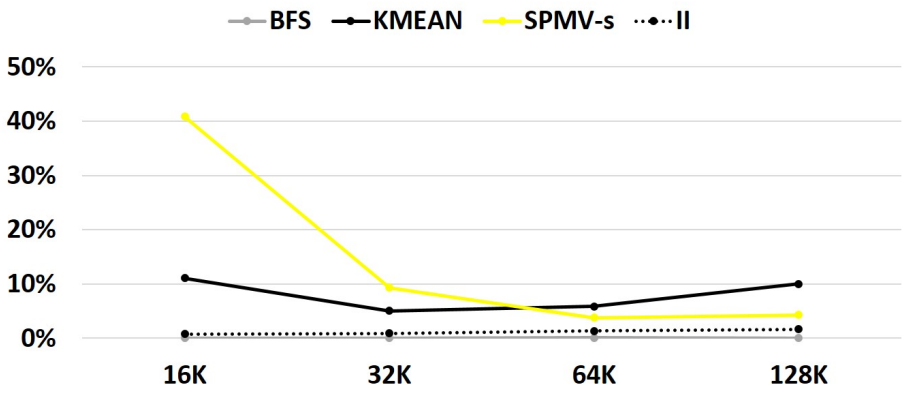

Figure 3.7: The ratio of throttling due to high memory divergence in LAWS-TH with varying L1 data cache size

LAWS-TH vs. LAWS-NMD: Comparing LAWS-NMD to LAWS-TH reveals the effect of throttling by a high memory divergence. A high memory divergence can severely decrease the degree of intra-warp locality in the L1 data cache since it can evict multiple cache blocks from a single warp's memory instruction. LAWS-NMD's performance is close to LAWS-TH's performance in BFS and II. However, LAWS-TH provides a substantially higher performance than LAWS-NMD in KMEAN with 16K and SPMV-s with 16K, 32K, and 64K due to the high memory divergence.

Figure 3.7 shows the ratio of throttling by a high memory divergence in LAWS-TH. The ratio is close to 0% in BFS and II. Thus, there is no noticeable performance difference between LAWS-NMD and LAWS-TH. In KMEAN with 16K, LAWS-NMD reaches only 68% of LAWS-TH performance due to the 10% ratio. In KMEAN with larger than 16K, however, LAWS-NMD reaches LAWS-TH's performance because larger cache size diminishes most of KMEAN's high memory divergence effect. The ratio decreases from 42% to 3% in SPMV-s as the cache size increases. Thus, LAWS-TH provides a significantly higher performance than LAWS-NMD in SPMV-s at 32K and 64K cache sizes. Especially, LAWS-NMD shows no performance improvement in SPMV-s with 16K since no intra-locality is detected due to a high memory divergence.

LAWS-NMD vs. CCWS: Both LAWS-NMD and CCWS might not properly throttle the

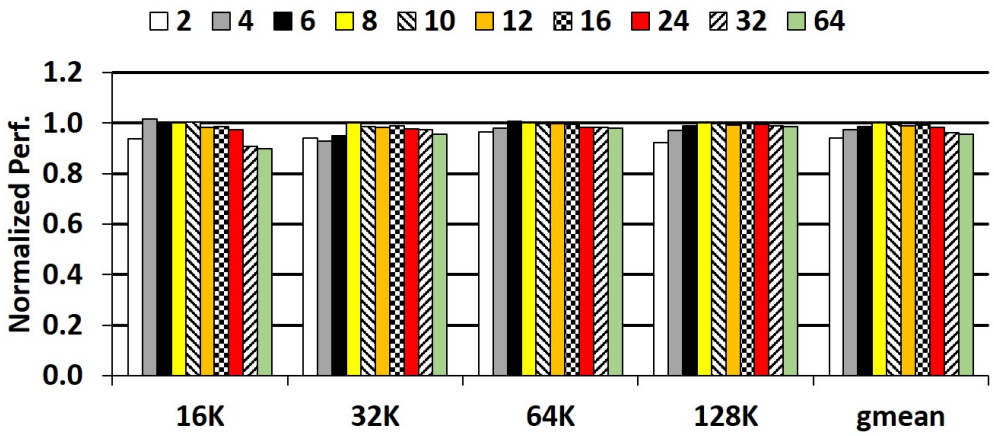

Figure 3.8: Performance of LAWS-TH with varying the stabilization period normalized to 8

number of warps when high memory divergence severely affects the intra-warp locality in the L1 data cache. Thus, CCWS shows low performances in KMEAN with 16K and SPMV-s with 16K similar to LAWS-NMD. However, LAWS-NMD provides a higher performance than CCWS in SPMV-s with larger than 16K. Furthermore, LAWS-NMD provides a higher performance than CCWS in BFS. CCWS's lower performance than LAWS-NMD comes from its reactive mechanism. CCWS starts throttling when it detects intra-locality loss in the victim tag [ROA12]. However, LAWS-NMD (also LAWS-TH) can actively launch throttling before the intra-warp locality is lost.

3.5.3.2 Sensitivity to stabilization period

LAWS-TH uses the average of Min_CNT and Max_CNT as the number of warps for scheduling once they are stabilized. We use the number of throttling periods, which does not update Min_CNT and Max_CNT as the average stabilization metric. If LAWS-TH takes the average too early, then the inaccurate average can lead to a sub-optimal performance. On the other hand, if it takes the average too late, then this long stabilization period can also result in a sub-optimal performance. Figure 3.8 presents the overall LAWS-TH performance in different L1 data cache sizes by varying the stabilization period normalized to 8 that shows the best performance. LAWS-TH shows a stable performance in the range of 6 to 24 across all cache

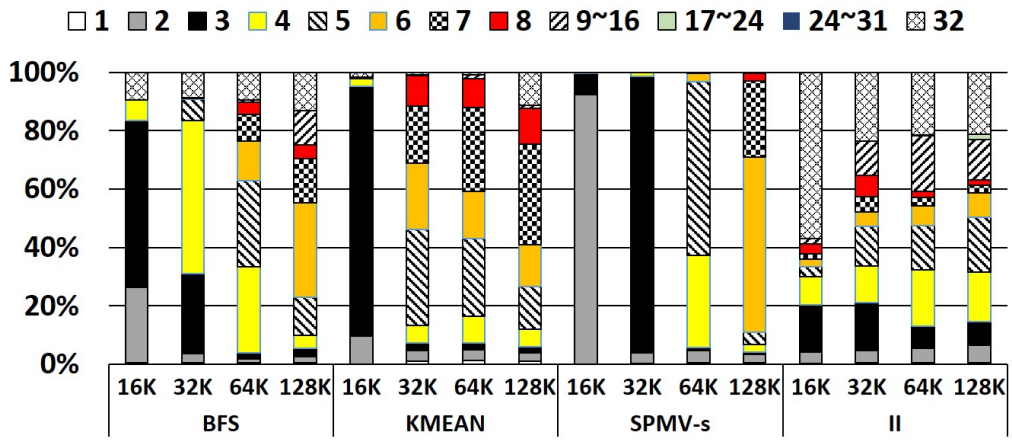

Figure 3.9: The execution time ratio of different number of warps in LAWS-TH with varying L1 data cache size

sizes. We choose 8 that shows the best performance for the stabilization period.

3.5.4 LAWS-TH Analysis

Figure 3.9 shows the execution time ratios of different numbers of warps in the LAWS-TH by varying the L1 data cache size. Overall, the number of warps in the throttling increases as the cache size increases in the throttling. In II, large numbers of warps (>7) are executed in a large portion of the execution time (about 40%). This comes from the fact that II consists of 17 different kernels which have different intra-warp locality degrees and different maximum numbers of warps per SM.

LAWS-TH uses the ceiling average of MIN_CNT and Max_CNT as the number of warps for throttling. Figure 3.10 shows the weighted average of the measured Min_CNT and Max_CNT values during the entire execution. Min_CNT's weight average stays from 1 to 3 across all cache sizes. On the other hand, Max_CNT's weighted average significantly increases as the cache size increases. Still, Max_CNT's weighted average stops increasing when the cache size is enough to hold most of intra-warp locality without throttling: 64K and 128K in KMEAN, and 128K in II.

Once throttling is enabled, LAWS-TH keeps throttling at a minimum of 1K cycles until

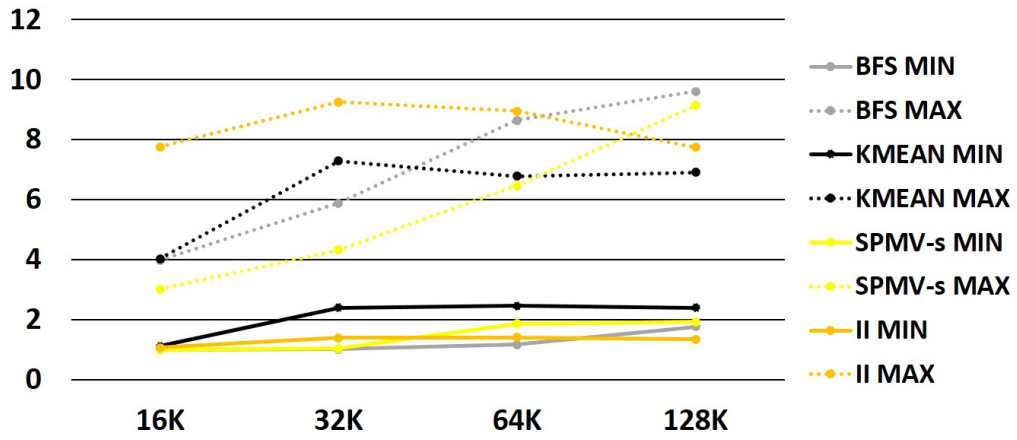

Figure 3.10: The weighted average of Min_CNT and Max_CNT in LAWS-TH with varying L1 data cache size

the MSHRs become full. Thus, the length of the throttling period depends on the benchmark's memory accesses per instructions and the cache miss rate. The L1 data cache accesses per instructions are 0.08, 0.09, 0.12, and 0.11 in BFS, KMEAN, SPMV-s, and II respectively. The miss rates of the L1 data cache are shown in Figure 3.12. The cache primary miss rate decreases as the cache size increases. Figure 3.11 shows the ratio of various throttling time lengths from the entire throttling time. Overall, the throttling time length increases as the cache size increases. Most throttling lengths are less than 128K cycles except for SPMV-s with 32K, 64K, and 128K cache sizes.

3.5.5 Implementation Complexity

The throttling unit (THU) is the main component of LAWS-TH. It borrows the locality detection units in the L1 data cache and MSHRs from the LAWS. We believe the implementation cost of the THU is negligible since they require only a few bytes of storage and few logic gates.

The implementation cost of THU is far smaller than previous works since LAWS-Th measures the intra-warp locality and the memory divergence on a warp base, not on an individual memory access. For example, we use a 5-bit WID in the L1 data cache for

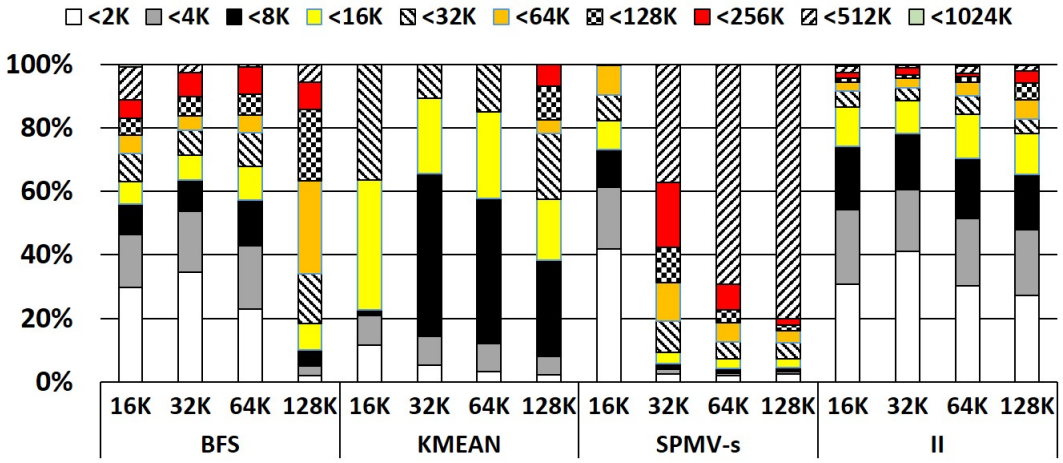

Figure 3.11: The ratio of different throttling time length ranges in LAWS-TH with varying L1 data cache size

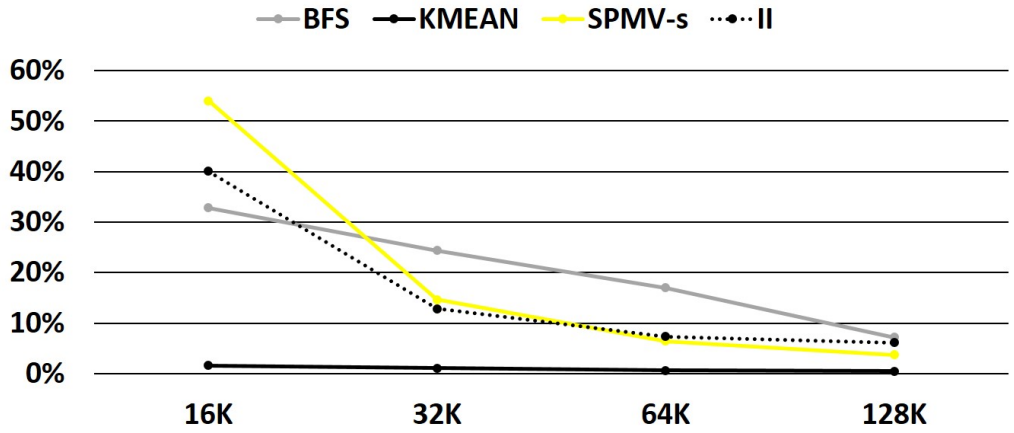

Figure 3.12: The miss rate (primary miss) of L1 data cache in LAWS-TH with varying cache size.

locality detection similar to CCWS [ROA12]. However, CCWS adds the victim tag array to detect intra-warp locality loss. Each tag size is 40 bits, and each warp has 16 tag entries. Thus, the total storage cost of the victim tag array is 2,560 bytes per core. However, our throttling mechanism requires only a single bit for each warp's intra-warp locality and four 5-bit registers: Hit_CNT, Min_CNT, Max_CNT, and Load_CNT. Furthermore, LAWS-TH simply measures the memory divergence by measuring the number of warp issuing memory requests when the MSHRs are fully consumed. Compared to DAWS [ROA13], LAWS-TH does not need a compiler support to predict each warp's data footprint, and it does not require the dynamic PC-based load instruction analysis at runtime that requires 5% of the core (SM) area [ROA13]. We believe that the implementation cost of the warp throttling should be minimized since they are effective only in a small number of benchmarks (4 out of 42 in our experiment).

3.6 Related Work

Guz et al. [GBK09] shows the "performance valley" where more threads do not always increase performance due to contention in a shared resource. Suleman et al. [SQP08] presents a feedback driven thread scheduling scheme to improve the performance by scheduling the optimal number of threads in CMPs. Cheng et al. [CLL10] propose a thread throttling mechanism to reduce the memory latency in multi-threaded CPUs.

Rogers et al. [ROA12] introduce cache-conscious wavefront (warp) scheduling (CCWS) that limits the number of active warps to prevent intra-warp locality loss in the L1D cache. CCWS uses a victim tag array to detect intra-warp locality loss, and reduces the number of active warps to retain more intra-warp locality. Divergent-aware warp scheduling (DAWS) is also proposed by Rogers et al. [ROA13]. DAWS proactively controls the number of active warps based on the prediction of each warp's data footprint with a compiler support. Instead of directly measuring the intra-warp locality loss like CCWS or the memory divergence like DAWS, our throttling mechanism reduces the intra-warp locality loss by the warp-

based indirect measurement of the degree of intra-warp locality and the degree of memory divergence. In contrast to our work, CCWS and DAWS do not explore the inter-warp locality loss by unfair scheduling policies. Regarding to the performance of LAWS-TH, we do not directly compare the performance of LAWS-TH against DAWS. However, we believe that the performance of LAWS-TH is close to DAWS based on the performance results against CCWS and B-SWL.

3.7 Conclusion

This work evaluates the performance of various throttling mechanisms in GPGPU applications. It explains how throttling reduces the intra-warp locality loss. Based on this observation, this study devises a throttling mechanism with little cost to improve the performance of GPGPUs further by reducing the intra-warp locality loss. In this work coupling with the LAWS, we achieve both locality-warp adaptive scheduling and throttling mechanisms with the minimum cost by sharing the functionality of the locality detection unit and exploiting the existing MSHR resources.

CHAPTER 4

Hardware Prefetching on GPGPUs

This chapter presents an effective hardware prefetching for general-purpose graphics processing units (GPGPUs). GPGPUs use hardware multithreading as a primary method to deal with memory latency. Prefetching is another technique to hide memory latency by overlapping memory accesses with computations. Thus, GPGPUs' memory latency hiding degree can be increased by prefetching. In current GPGPUs, two software prefetching methods are available. First one is software prefetching using temporary variables [RRS08]; however, it increases the instruction count and register usage per thread. This increased register usage can seriously limit thread-level parallelism [YXK10]. Software prefetching instructions added in NVIDIA's Fermi GPGPUs perform prefetching without using temporary variable; still, it increases instruction count.

Hardware prefetching can avoid such overheads of software prefetching by generating prefetch requests without temporary varibles or instructions. Various hardware prefetching mechanisms for conventional CPU systems were proposed in the literature [CB95, FPJ92, HM94, ISK04, JG97, Jou90, NS04, PK94, NDS04, SKT05]. However, there are some challenges to apply conventional prefetching mechanisms for GPGPUs due to microarchitectural differences between CPUs and GPGPUs [LLK10]. On the other hand, GPGPU's microarchitectural differences from CPUs gives us chances to improve prefetching mechanisms by exploiting the intrinsic characteristics of GPGPUs. Based on these observations, we propose simple hardware prefetching mechanism that overcome the challenges and utilize GPGPU's unique microarchitectural feature. We also propose an efficient prefetch throttling mechanisms by exploiting GPGPU's thread block execution mechanism. Our prefetching and

throttling improves the performance 3% to 14% in different dram latency configurations.

4.1 Introduction

This section introduces conventional PC-based stride hardware prefetching mechanism that is used in our basic prefetching scheme. It also explains the challenges to apply it for GPGPUs.

4.1.1 PC-based Stride Prefetching Mechanism

Memory access patterns in GPGPU applications are usually more regular than CPU applications due to GPGPUs' single-instruction multiple-thread (SIMT) model. All threads are execution same instructions in SIMT model, and memory addresses are usually specified by thread ID or based on thread ID in most of GPGPU applications. In other words, each thread has usually the same memory access pattern though each thread accesses different memory regions in GPGPU applications.

Because of GPGPU's common regular memory access patterns, unit and non-unit stride hardware prefetching mechanisms can be suitable. However, non-unit stride prefetcher is preferable since it can cover both unit and non-unit stride patterns. We choose PC-based stride prefetcher [CB95] as a basic scheme for our prefetching mechanism.

The PC-based stride prefetcher detects repeated stride among memory instructions having the same PC value. The prefetcher has a PC stride table which is accessed and updated by all memory operations. Figure 4.1 shows the state transition graph. Two bits are used to represent the state (0:INIT, 1:TRANSIENT, 2:STEADY, 3:No-Predict). When a memory instruction is issued, an entry for the memory instruction is searched with its PC value. If the entry does not exist, the memory operation fills one entry with its PC value, its memory address, the default stride value(0), and initial state (INIT). If the entry is found, a stride value is calculated by subtracting the prev_addr in the entry from the memory address of the current memory operation. This current stride value is compared to previous stride value

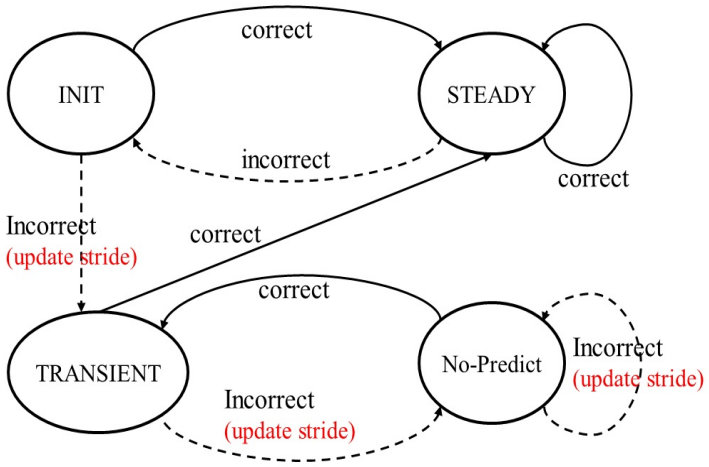

Figure 4.1: State Transition Graph

in the entry. If both of strides matches, the current stride is considered a correct value; otherwise, it is considered a incorrect value. Then, the state in the entry transits depending on the current state and the result of the stride comparison shown in figure 4.1, and the stride value is also replaced with the current stride if the current stride value is incorrect except for the transition from STEADY to TRANSIENT. When the state is TRANSIENT or STEADY, prefetching is issued with the prefetching address (prev_addr + stride).

Figure 4.2 shows an example code and the PC-stride table entry in first three iterations. The table was empty before the first iteration in the example. After the first iteration, three entries are fill with zero stride and INIT state. Then stride(4) is detected in second iteration, and the prefetcher starts to send prefetching requests with a prefetching address. Finally, the all states are changed into STEADY state in the third iteration since the strides of second and third iteration matches

4.1.2 Challenges of Hardware Prefetching on GPGPUs

There are some challenges to apply a hardware prefetching to GPGPUs [LLK10]. First, the chances of performing hardware prefetching in GPGPUs is much less than in CPUs because the number of instruction executed by a thread in GPGPUs is usually much less than in

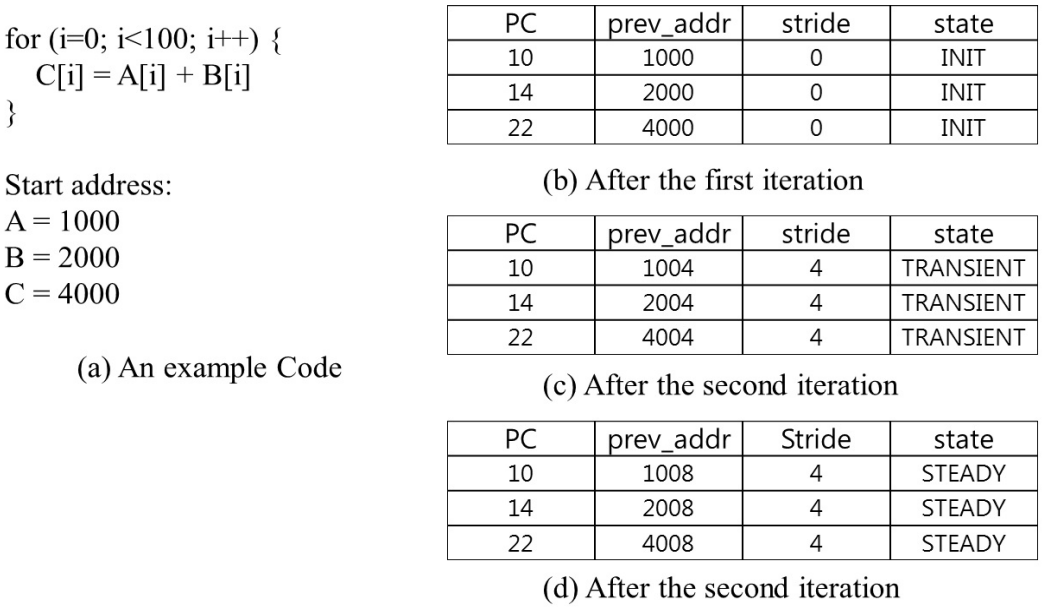

Figure 4.2: An example of PC-stride prefetching

CPUs due to GPGPUs' thread-level parallelism. For example, a CPU codes use a loop for a vector addition that performs one addition at each loop iteration. Thus, each iteration can perform the prefetching for next iteration's data. However, there is no loop in a GPGPU code for the vector addition because all the additions are distributed to all the threads. Each thread performs one addition of the vector addition independently in GPGPUs.

Another problem is that memory access pattern of a warp can be interfered by other warps' memory access by warp interleaving in GPGPUs. The conventional PC-stride prefetcher calculates the stride value by checking the address of memory requests. However, this calculation can result in a wrong stride value although all the warps have the same memory access pattern due to warp interleaving.

To avoid the memory access pattern interference, warp-ID tag can be added to PC-stride prefetcher. Then, PC-stride with warp-ID tag can detect each warp's stride correctly. However, it increases the size of the stride table since it has to include each warp's stride values. Since the maximum active number of warps in each GPU core is from 32 (in our configuration). This huge size of the stride table is not practical for GPGPUs [LLK10]

PC	Address	Stride	State	#Access
30	306333824	245760	STEADY	25
31	322333824	245760	STEADY	25
32	338333824	245760	STEADY	25
...	...	...	...	...

Table 4.1: A snapshot of the Stride Table in PCST

4.2 Hardware Prefetching for GPGPUs

To overcome the challenges described in section 4.1.2, our prefetching mechanism exploits the intrinsic characteristics of GPGPUs, single-instruction multiple-thread (SIMT). All thread usually has a similar memory access pattern in GPGPUs due to SIMT. Based on this observation, the prefetching scheme only measures the memory access pattern of only single thread, then it applies the detected pattern to all other threads. This single thread based PC-stride prefetching solves the memory interference problem by tracking of only single thread's memory requests. Furthermore, it does not require the large stride table for all the warps.

We also use thread block execution mechanism to increase prefetching opportunities in GPGPUs. A kernel in GPGPUs consists large number of thread blocks, and only few thread blocks (up to 8) can be executed concurrently by a SM. Memory access patterns of all threads can be similar to each other due to SIMT model. It leads to the fact that all thread block's memory access patterns can be also similar to each other since thread block is just a group of multiple threads. We exploits the thread block execution mechanism to build prefetcher to detect patterns between thread blocks.

The section describes our single thread based prefetching within thread block, and it explains the prefetching mechanism across thread blocks. Finally, it explains our thread block based prefetch throttling mechanism.

4.2.1 Single Thread Prefetching Extension within Thread Block

To detect strides in repeated memory access patterns like loop iterations, we use PC-based stride prefetching mechanisms. However, our PC-based stride prefetcher keeps tracks of memory access patterns of only single thread (thread ID=0), then apply this pattern to all other threads for prefetching. The thread ID is used for the isolation of single thread's memory accesses from other threads, but the stride table does not have thread ID information. The mechanism of PC-based stride prefetcher using single thread (PCST) is identical to conventional PC-based stride prefetcher except that the stride is detected and updated by single thread and used by all other warps.

Table 4.1 shows a snapshot of the stride table of PCST from one of benchmarks. The table of PCST is similar to the table of the conventional PC-based stride table except for one column to count the number of accesses. The number of accesses is used in our another prefetching scheme, inter-PC stride prefetching discussed in section 4.2.2. All entries in the table present three different load operations with initial stride value of 245,670 with the STEADY state. All other warps issue prefetching when each warp performs load operations with the same PC values (30,31,32). All the stride values in this example are same by chance, and it shows one example of regularity of memory accesses in GPGPU applications.

4.2.2 Single Thread Prefetching Extension across Thread Blocks

Our PCST cannot detect memory access pattern in applications not having any loops. Table 4.2(a) shows the entries of the stride table of PCST when a thread block finishes from one of our benchmarks. In this table, all memory instructions are executed once due to the lack of any loops, so there is no stride detected. However, prefetching can be issued for next instruction in this case if there is a constant pattern between memory instructions. So, such prefetcher can be built by comparing the stride between memory instructions among threads. Still, the chances to apply such prefetching are very little since the progress rate of each thread is similar in GPGPUs. When the stride is detected, most of threads probably

PC	Address	Stride	State	#Access
20	268439552	0	INIT	1
22	268439616	0	INIT	1
24	268439680	0	INIT	1
26	268439744	0	INIT	1
28	268439808	0	INIT	1
30	268439872	0	INIT	1
32	268439936	0	INIT	1
34	268440000	0	INIT	1
50	268440064	0	INIT	1

(a) The stride table of PCST

PC	Stride	State
20	64	INIT
22	64	INIT
24	64	INIT
26	64	INIT
28	64	INIT
30	64	INIT
32	64	INIT
34	64	INIT

(b) Inter-PC stride table

Inter-PC stride calculation:
Stride = Address of next entry – Address of current entry (in the stride table of PCST)
* Inter-PC stride considers entries having only one access

ex) PC:20, stride = 268439616 (PC:22) – 268439552 (PC:20) = 64
PC:22, stride = 268439680 (PC:24) – 268439616 (PC:22) = 64

(c) Inter-PC stride value calculation

Table 4.2: Updating inter-PC stride table when a thread block finishes

finishes their corresponding memory instructions without any loops. Therefore, it is not efficient to apply such prefetcher within a thread block execution.

We propose inter-PC stride prefetching mechanism (ITPC) to deal with this problem. It checks memory accesses per thread block basis. When a thread block finishes, ITPC scans the stride table of PCST and calculates the stride between two addresses of two entries that have only one access. Table 4.2 shows the updating process of the stride table in ITPC. Each entry in the PC-based stride table has only one access (table 4.2(a)) after a thread block finishes, so no stride is detected. ITPC scans the stride table of PCST and calculates the address differences between each two entries having only one access as shown in table 4.2(c). Each entry in inter-PC stride table has 64 stride value with INIT state in the inter-PC stride table after the update as shown in table 4.2(b). When another block finishes later, these stride between memory instruction are calculated again and compared to the stride values in the table. The state transition of the inter-PC table is identical to conventional PC-based stride prefetcher. All stride value in the inter-PC stride table are also same by chance in this

example.

ITPC tries to detect memory access pattern across thread blocks, but examines single thread's memory access patterns of each thread block. The progress of issuing prefetching in ITPC is identical to conventional PC-based stride prefetcher though the stride detection and update mechanism is different. ITPC is accessed only if prefetching is not issued from PCST. When a prefetching is not issued from PCST, ITPC checks the memory request's PC value and search the corresponding entry with the PC value in the inter-PC stride table. If the entry is found, prefetching is issued using the stride value when the state in the entry is TRANSIENT or STEADY. Still, the request does not update the entry. The entries are updated when ITPC scans the stride table of PCST.

If the size of PCST stride table is large, the cost of scanning over the PCST stride table can be expensive. However, we consider it as an acceptable cost due to two reasons. Firstly, the cost of the scan is small in most of cases since the size of PCST stride table is usually small; the table stores only single thread memory access patterns, and the number of load operation per thread in GPGPU applications is much less than in CPU application because of thread-level parallelism. Secondly, the frequency of scan is also quite small since scanning over the PCST stride table is done in the granularity of thread block execution. Therefore, the number of scans is same as the number of thread blocks which is quite a small number.

4.2.3 Thread-Block basis Prefetch Throttling

Prefetch throttling can be improved by exploiting thread block execution mechanism. Conventional prefetch throttling mechanisms usually uses some interval based on heuristic to measure the effectiveness of prefetching. Thread block execution provides a good interval to measure the effectiveness of prefetching since each thread block execute the entire instructions of the application.

It can provide find-grain control of prefetch throttling because the throttling interval is decided by an application. Moreover, we can decide the effectiveness of prefetching very

```
While(#thread blocks to execute > 0)
{
  if(instruction==load and thread id==0)
  {
    // send PC and addr to PCST stride table
    update_PCST(PC, addr);
}

  if(any thread block finishes)
  {
    // update ITPC stride table
    update_ITPC();
    if(#useful prefetching < #useless prefetching)
    {
      Remove all entries in both of stride tables
      prefetch_throttling++;
    }
    if(prefetch_throttling>2)
      Turn off prefetcher
}
```

(a) Pseudo Code: update prefetcher's states

```
While(#thread blocks to execute > 0)
{
  stride=0;
  if(instruction==load)
    stride = PCST(PC, addr);
  if(stride==0)
    stride = ITPC(PC);
  if(stride!=0)
    issue_prefetching(addr+stride);
}
```

(b) Pseudo Code: issuing prefetching request

```
update_PCST(PC, addr)
{
  search the entry with PC
  if(the entry is found)
    update the entry with PC and addr
  else
    add new entry in PCST stride table
}

update_ITPC()
{
  scan the PCST stride table
  calculate stride between two entries
  search the entry in inter-PC stride table with PC
  if(the entry is found)
    update the entry with PC and stride
  else
    add new entry in inter-PC stride table
}
```

(c)Pseudo Code: update stride tables

```
int PCST(PC, addr)
{
  stride = 0;
  search the entry with PC
  if(the entry found)
    if(state in the entry == TRANSENT or STEADY)
        stride = the stride value in the entry
  return stride;
}

int ITPC(PC)
{
  stride = 0;
  search the entry with PC
  if(the entry found)
    if(state in the entry == TRANSIENT or STEADY)
      stride = the stride value in the entry
  return stride;
}
```

(d) Pseudo Code: stride retrieval from prefetcher

Figure 4.3: Pseudo Code

quickly by measuring prefetching accuracies of few blocks since all thread block will show similar prefetching accuracy due to the characteristics of SIMT model. For example, prefetching can be turned off quickly when the accuracy of prefetching is low in few blocks.

Our prefetching throttling mechanism compares the number of useful prefetchings and the number of useless prefetchings whenever a thread block finishes. If the number of useless prefetching is larger than the number of useful prefetching, prefetcher resets both of stride tables and restart the memory pattern detection. When the reset occurs more than twice, prefetching is completely turned off.

To summarize, our prefetching mechanism, single thread prefetching extension (STPE) consists of two stride prefetcher(PCST and ITPC) and prefetcher throttling logic. Figure 4.3

Number of Cores	14 Streaming Multiprocessors with 8 SIMD Width
MAX # of threads / Core	**768** / 1024, up to 8 thread blocks
Core specification	900 Mhz, in-order scheduling, 4 pipeline stage
On-chip Storage	Scratchpad memory: **16**/32/64 (KB) (16 banks) Constant Cache: 8KB (2-way set assoc. 64B lines LRU) Texture Cache: 64KB (2-way set assoc. 64B lines LRU) L1: 16 KB (8-way set assoc. 64B lines LRU)
Memory Controller	out of order (FR-FCFS) scheduling [RDK00]
DRAM	2KB page, 16 banks, 8 channel, 57.6 GB/s bandwidth tCL = 11, tRCD=11, tRP=13, BurstLength=8
Interconnection	Fixed latency: **50** / 100 / 200 / 400 cycles

Table 4.3: Simulator Configuration (**bold**: baseline configuration)

shows the overall prefetching processes of STPE in pseudo code. In contrast to conventional stride prefetcher, the procedure of updating stride tables is separated from the procedure of issuing prefetching request since the subjects of updating stride tables (thread ID=0 and PCST stride table) are different from the subjects of accessing prefetcher (all warps).

4.3 Methodology

We use the GPGPU-Sim simulator v2.x [BYF09], which is based on NVIDIA's CUDA programming model [NVI11b]. Table 4.3 shows the simulation configuration which is close to NVIDIA's 8800GT [NVIb]. Additionally, it can simulate some features of NVIDIA's Fermi architecture since it has configurable L1 data cache and all other on-chip storages, such as, scratchpad memory, register files and texture & constant caches. One of important features of the simulator used in our prefetcher is inter-warp memory coalescing. The goal of inter-warp coalescing is to reduce memory traffic by blocking sending read memory requests if the same memory requests are already in progress. This inter-warp coalescing is extended

Benchmark	Abr.	Suite	Grid Dimension	Thread Block Dimension	Total Threads
Black-Scholes option pricing	Black	SDK	(480,1,1)	(128,1,1)	61440
Convolution Separable	Conv	SDK	(4,128,1) (32,8,1)	(16,4,1) (16,8,1)	32768 32768
Mersenne Twister	Merse	SDK	(32,1,1) (32,1,1)	(128,1,1) (128,1,1)	4096 4096
MonteCarlo	Monte	SDK	(128,1,1) (256,1,1)	(128,1,1) (256,1,1)	16384 65536
ScalarProd	Scalar	SDK	(128,1,1)	(256,1,1)	32768
StreamCluster	Stream	Rodinia	(128,1,1)	(512,1,1)	65536
Backprop	Backprop	Rodinia	(1,2048,1) (1,2048,1)	(16,16,1) (16,16,1)	524288 524288
Breadth First Search	BFS	Rodinia	(128,1,1)	(512,1,1)	65536
CFD	CFD	Rodinia	(1212,1,1)	(192,1,1)	232704

Table 4.4: Benchmark Properties (CFD benchmark has six kernels)

to prefetching requests to prevent sending on-demand memory requests if the same request issued by prefetcher is still in progress and vice-versa.

We evaluate our prefetching mechanism with nine memory-intensive benchmarks taken from CUDA SDK [NVIa], Rodinia [CBM09b]. Table 4.4 shows the list of our benchmarks with each benchmarks property, such as the number of threads per block, the number of thread blocks, and total number of threads in a kernel

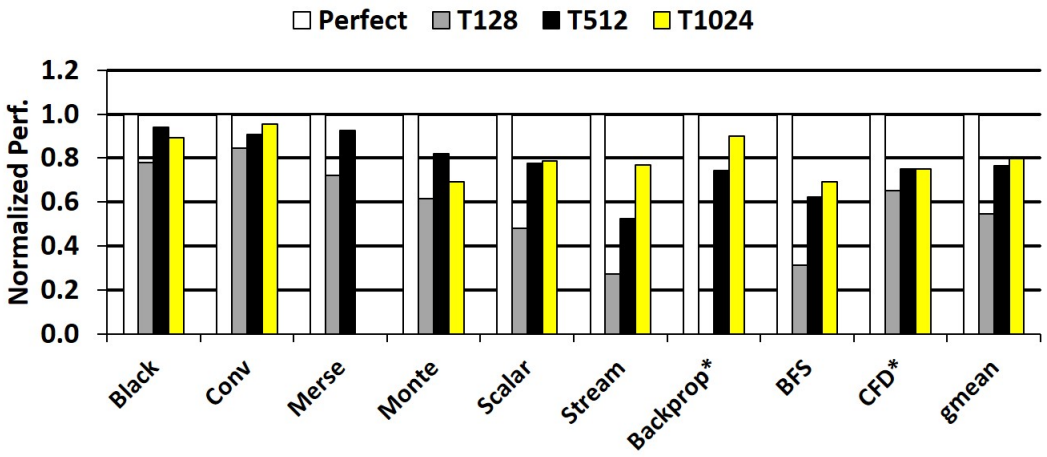

Figure 4.4: Performance of different number of threads per SM
(Perfect: always cache hit, T#= # threads per SM

4.4 Experiments

In this section, we shows our simulation results and evaluations. First, we evaluate GPGPU's performance with different degrees of thread-level parallelism to understand the upper-bound of performance improvement by prefetching. Second, we evaluate our prefetching mechanisms with various configurations and compare it to previous work(MT-HWP) [LLK10]. Finally, we measure the performance impact of our prefetching mechanism with regard to different memory latency.

4.4.1 Hardware Multithreading Performance

The motivation of prefetching in GPGPUs is to hide memory latency that not completely hidden by hardware multithreading due to not enough thread-level parallelism. Therefore, there is little room to improve performance by prefetching if there are enough number of threads to hide memory latency by hardware multithreading.

Our first experiment is to measure the performance improvement by increasing the degree of hardware multithreading. To change the degree of hardware multithreading in the experiment, we limit total number of threads per SM (128, 512, 1024) by changing the maximum

number of thread blocks per SM (1, 4, 8). So, the number of threads per a thread block is 128 in this experiment. As shown in table 4.4, however, each benchmark has different number of threads per a thread block (thread block dimension). We modify the thread block dimension of some benchmarks for this purpose except for Backprop benchmark(256) and CFD benchmark(192); the thread block dimension of these two benchmark is difficult to modify since the modification can affect the benchmark's algorithm.

Figure 4.4 shows that performance (IPC) increases as the number of threads per SM is increased in most of cases. The performance gap between perfect memory model (the first bar) and different number of threads per SM (T128, T512, and T1024 with default memory model(No cache)) presents the *upper-bound* of performance improvement by prefetching. This gap decreases as the number of threads per SM increases. Therefore, this experiment confirms that the benefit of prefetching decreases as thread-level parallelism(the degree of hardware multithreading) increases.

In Black benchmark and Monte benchmark, however, the performance is decreased with 1024 threads (8 thread blocks) in the figure 4.4 due to *thread block imbalance* [BYF09]. When the number of thread blocks assigned to each SM is different, it is called thread block imbalance. For example, there are six thread blocks in a kernel, and two SMs. Each SM can execute two thread blocks concurrently. In the beginning, four thread blocks are executed by two SMs, and two thread blocks are left. If each SM finishes the execution of two thread block at the same time, one thread block assigned to each SM. Let the execution time of each thread block is T, then the total execution time is 3T in this case. However, if one of SM finishes the execution of first two thread blocks earlier than the other one. Remaining two thread blocks are assigned to the SM that finishes the execution early. Soon, the other SM finished the execution, but it becomes idle since there is no thread block to execute. Therefore, the total execution time is 4T in this thread block imbalance case. The thread block imbalance problem worsens as the maximum number of thread blocks per SM increases.

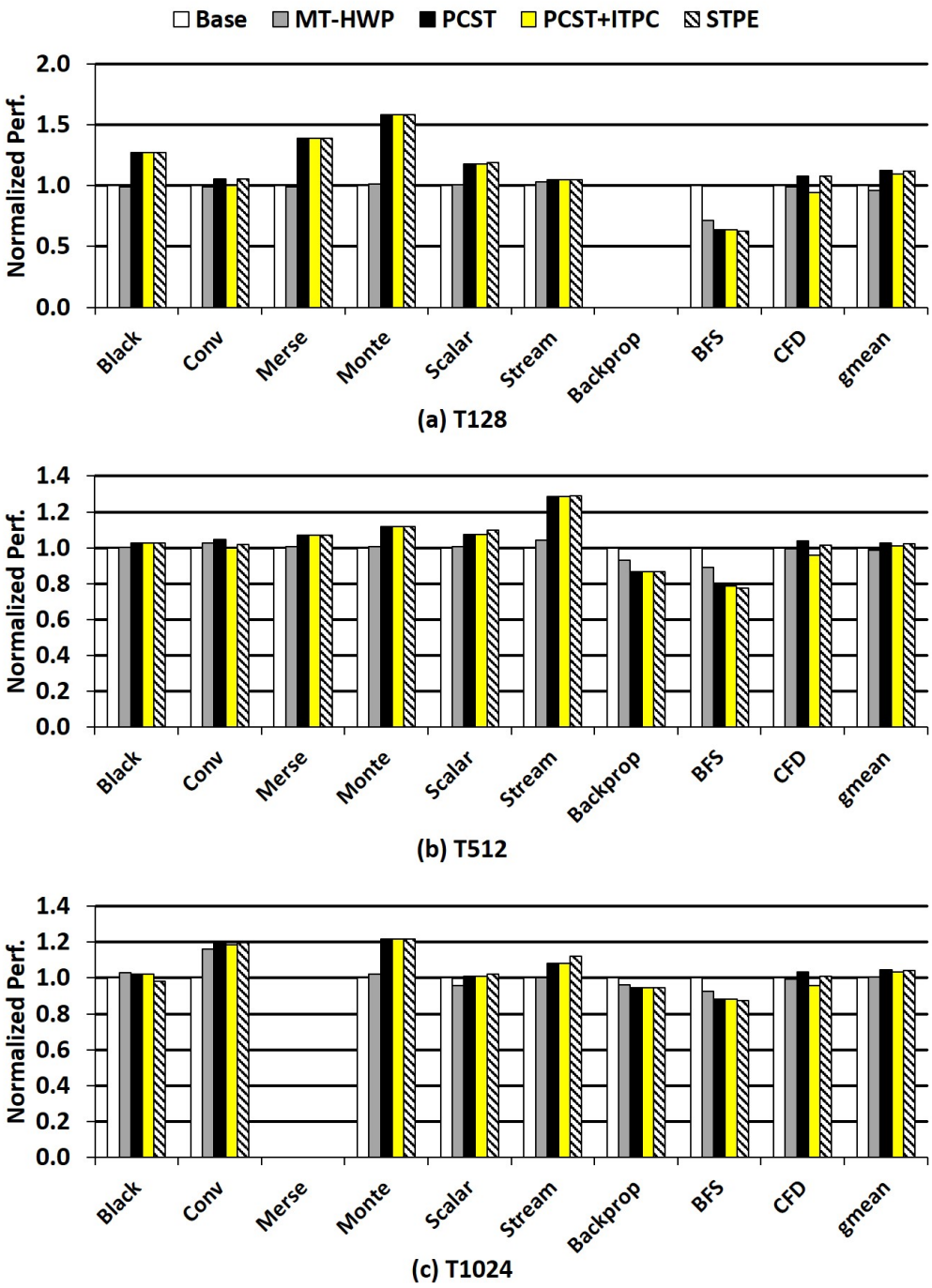

Figure 4.5: The Effectiveness of prefetching with different number of threads per SM (MT-HWP:Per-warp training and inter-thread prefetching [LLK10], STPE: PCST+ITPC+throttling)

4.4.2 Evaluation of Single Thread Prefetching Extension

To evaluate our prefetching mechanism, we run simulation with five configuration: L1 data cache, MT-HWP prefetching(inter-thread prefetching and per-warp training with stride promotion [LLK10]), PCST, PCST+ITPC, and STPE (PCST+ITPC+Throttling). L1 data cache is used for a prefetch cache in all of four prefetching configurations, and demand fetches are not loaded into the prefetch cache. We also simulate those five configurations with different number of threads per SM.

Figure 4.5 shows the normalized performance of five configuration relative to baseline configuration (No cache) with different number of threads per SM. Our prefetching increases performance other 7 benchmarks from 9 benchmarks except for Backprop and BFS. Only cache increase performance in Backprop and BFS. The number of prefetching in these two benchmarks is much less than other benchmarks since it is difficult for prefetcher to detect memory access patterns in these two benchmark.

As we discussed in previous section, the benefit of prefetching decreases as the number of threads per SM increases. The average speed-up by our prefetching mechanism (PCST+ITPC+throttling) are 24.4% with 128 threads, 9.1% with 512 threads, and 7.3% with 1024 threads.

Conv and CFD benchmarks have no loop involving load memory operations. Therefore, the PCST does not improve performance in these two benchmarks, but the ITPC does. When prefetch throttling is applied, the performance is not changed significantly in most of cases. However, it cannot remove the negative effect of prefetching in Backprop and BFS benchmarks since the number of generated prefetching is only few percentage of total memory requests without throttling. So, in these benchmarks, throttling reduces the number of prefetching further, but it is not enough to increase performance.

We do not see a significant performance improvement with MT-HWP mechanism. Their inter-thread prefetching generates lots of late prefetching in our simulation due to the warp scheduling policy of GPGPU-Sim. GPGPU-Sim uses a round-robin scheduling policy that

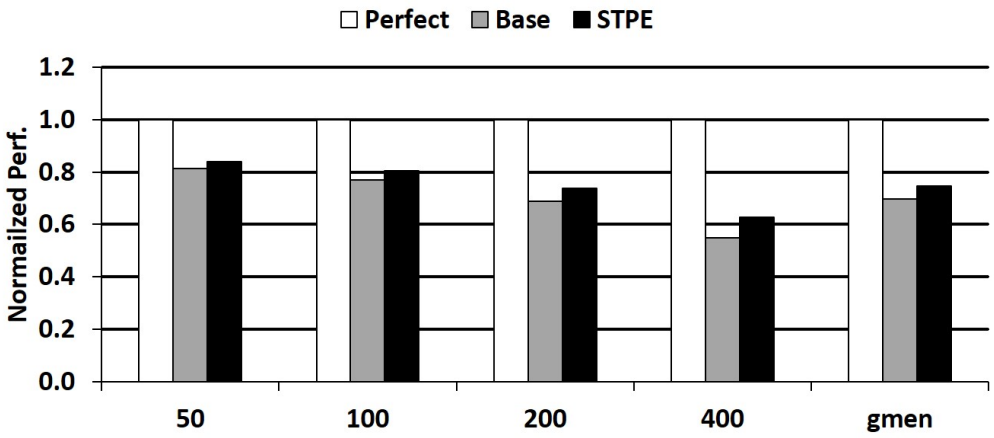

Figure 4.6: The simulation result with different memory latency
(Average Memory Latency = 30˜ 40 cycles [DRAM latency] + 2 x inteconnection latency)
(Perfect:(Always cache hit), STPE: PCST+ITPC+Throttling)

Benchmark	Black	Conv	Merse	Monte	Scalar	Stream	Backprop	BFS	CFD
#Thread/SM	512	384/384	512/640	640/512	512	512	512/512	512	192/576/768

Table 4.5: The number of threads per SM in the simulator's baseline configuration

switches a warp in every instruction. Thus, the time between prefetch and fetch is very short in our configuration. This leads the poor performance of MT-HWP.

4.4.3 Prefetcing vs. Memory Latency

So far, we simulate with unlimited on-chip storages to evaluate the prefetching performance with regard to thread-level parallelism. However, the number of threads per SM can be limited by the usage of on-chip storages such as register file and scratchpad memory. Furthermore, the number of threads per SM can be limited by other constraints (the maximum number of threads per SM(768, 1024, or 1568) or the maximum number of thread blocks(8)).

Table 4.5 shows the number of threads per SM running concurrently in the simulator's baseline configuration that close to NVIDIA's Telsa architecture. In this section, we simulates benchmarks in this baseline configuration without changing thread block dimension of benchmarks. We only change the memory latency to measure performance improvement by

prefetching with regard to memory latency. The memory latencies are configured by changing the interconnection network latency. Figure 4.6 shows the simulation result with various interconnection network latencies. Our prefetching (PCST+ITPC+Throttling) improves the performance of 7 benchmarks, and L1 cache improves performance in Backprop and BFS benchmarks in the baseline configuration. The prefetching improves the performance by 3%, 5%, 7%, and 14% in 50, 100, 200, and 400 of the interconnection latency respectively. These results shows that the performance improvement by prefetching increases as memory latency increases since more memory stall cycles can be hidden by prefetching.

4.4.4 Related Work

To deal with various data access patterns, lots of hardware prefetching mechanisms for CPUs are proposed in the literature. The simplest data access pattern is unit-stride access pattern that the distance of consecutive access is one unit (e.g., an access pattern of A[0], A[1], A[2],..., of array A). Jouppi [Jou90] proposed prefetcher using stream buffers for unit-stride pattern. Stream buffers are FIFO buffers, and each buffer has prefetched cache blocks of one stream. If cache misses, each head entry of all stream buffers are checked to see if any buffer has that block. If the block is found in a buffer, the block is fetched into cache, and prefetching issued to fill the last entry of the buffer.

Non-unit stride access pattern, the distance of consecutive accesses is larger than one unit (e.g., an access pattern of A[0], A[3], A[6],..., of array A), is more complex than unit stride pattern. Stride prefetchers [CB95, PK94] were proposed for non-unit stride pattern. PC-based stride prefetcher [CB95] calculates an address difference between memory operations having the same program counter (PC) value. If it detects a constant stride, prefetching is issued based on the stride value. Stride prefetcher without PC [PK94] was proposed for off-chip stride prefetching (e.g., prefetching in disk cache) since the PC values is only available in on-core prefetcher.

The most complex memory access pattern is non-stride memory access pattern such as a

linked list structure traversal or indirect array references. Some hardware prefetching mechanisms [HM94, JG97] were proposed for these irregular memory access patterns. However, most of prefetching mechanisms deal with regular memory access pattern due to design complexity for detection of irregular memory access patterns.

Lee et. al [LLK10] propose hardware prefetching mechanisms (MT-HWP) for GPGPUs. They claim three challenges to apply hardware prefetching in GPGPUs: less prefetching opportunities of GPGPU applications, memory access pattern interference by warp interleaving, and the division of prefetcher storage by multiple active warps. Then, they present three solutions for those challenges: inter-thread prefetching, per-warp training, and stride promotion. MT-HWP consists of inter-thread prefetcher and PC and warp ID based stride prefetcher. The inter-thread prefetcher has inter-thread stride table, and the PC and warp ID based prefetcher has per-warp stride table and global stride table for promotion. Both prefetcher detects stride values independently. The prefetcher check the stride tables in sequence of the global stride table, the inter-thread table, and per-warp stride table. For example, when stride value is found in global stride table, prefetching is issued and other tables are skipped. However, the performance of the inter-thread prefetching is sensitive to the warp scheduling policy which leads a poor performance in our evaluation. Furthermore, our STPE uses single thread that has a smaller size of the stride table than MT-HWP. Finally, our STPE does not require the stride promotion.

Prefetching is useless if prefetched block is evicted from the storage (e.g, cache or some buffer) without any usage. To prevent these useless prefetchings, some feedback mechanism [EML09, EMP09, SMK07] was proposed in the literature. Basically, these mechanism measures the accuracy of prefetching by comparing the number of useful prefetching and useless prefetching. Our prefetch throttling mechanism is also based on this scheme.

4.5 Conclusion

This study shows how conventional prefetching mechanism properly applies in GPGPUs exploiting the property of SIMT programming model. Although we use PC-based stride prefetcher for the basic scheme, other conventional prefetching mechanisms can be effectively applied in GPGPUs since memory access patterns of GPGPU applications are more regular and simpler than the ones of CPU applications. We also proposed a prefetching enhancement using thread block execution. Detection stride across thread blocks used in ITPC offers new prefetching opportunities in GPGPU systems. Thread-block basis prefetch throttling presents the fine-grain control of prefetching throttling.

4.6 Acknowledgement

I would like to thank to Professor Yuval Tamir for his suggestions and feedback on improving this study.

CHAPTER 5

Tag Shared Cache Miss Handling Architecture

General-purpose graphics processing units (GPGPUs) mainly rely on hardware multithreading to deal with a long memory latency. GPGPU's hardware multithreading requires an efficient cache miss handling architecture (MHA) to support non-blocking loads that can allow a GPGPU to continue executing instructions upon a load miss. The MHA stores the information of a load miss in the miss status holding registers (MSHR) to support non-blocking loads. However, the scalability of the MHA is limited due to the MSHR's fully-associative structure.

This paper presents a tag shared MSHR array (TSMA) that is sufficiently scalable to support GPGPU's massive hardware multithreading. The tags (cache block addresses) of all MSHR entries are removed in the TSMA. Instead of having a tag in an MSHR entry, an MSHR entry shares the tag of a cache block. The shared tag provides the MSHR entry index information. By this tag sharing, all the MSHR entries' tag comparisons upon an access to MSHR are eliminated. Due to the tag comparison elimination, the TSMA can have an array structure of MSHR entries that can easily increase the number of MSHR entries without high implementation cost and high power consumption caused by fully-associative structure.

TSMA, however, has two side effects. First, if all the cache blocks in a cache block set are used by tag sharing, then a cache miss on the cache block set locks the entire cache until one of shared tag is released. Second, the tag sharing by the TSMA has the same effect of evicting the cache block upon a cache miss that is supposed to be evicted when the cache block is filled. To relieve these side effects, this paper proposes a hybrid MSHR structure consisting of a small size of a conventional MSHR and a large size of TSMA. Our experiment

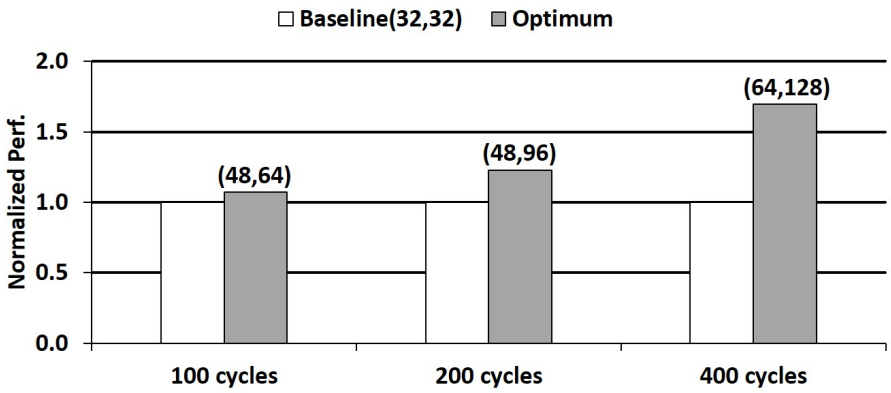

Figure 5.1: Performance of the optimal MSHR size over baseline (32, 32) MSHR by varying the minimum DRAM's latency in 18 MSHR sensitive benchmarks: #MSHR (L1, L2)

shows that the hybrid MSHR provides 99% performance of a conventional fully-associative MSHR in 28 benchmarks with only 33% and 59% of a conventional MSHR's power and area budget, respectively.

Figure 5.1 shows GPGPU's performance with the optimal MSHR size in each L1 data cache and each L2 bank in the L2 unified cache over the GPGPU-Sim [BYF09]'s baseline in 18 MSHR sensitive benchmarks. The baseline uses 32 MSHR entries in L1 and L2 bank caches. The performance improvements are 7%, 23%, and 70% in 100, 200, and 400 cycles of DRAM minimum latency, respectively. The optimal numbers of MSHR entries (L1, L2 bank) are (48, 64), (48, 96), and (64, 128) in 100, 200, and 400 cycles of the latency, respectively. We find the optimal MSHR size by the sensitivity experiment as discussed in section 5.4.1. This figure shows commonly used (32, 32) MSHR entries in L1 and L2 bank is not the optimal MSHR size. More MSHR entries are needed to achieve the maximum performance: 48/64 in L1, 64˜128 in L2 bank depending on the minimum DRAM latency. This result motivates us to devise scalable MSHR structure since conventional MSHR's fully associative structure limits its scalability.

MSHRs can be organized in a unified structure or multiple MSHR banks that each bank deals with non-blocking loads of each cache bank. A unified structure, however, even with 32 MSHR entries used in the baseline is unlikely feasible with conventional fully-associative

structure [TCT06]. On the other hand, banked MSHRs divide the total MSHR entries into each cache bank. Thus, banked MSHRs provide higher bandwidth and scalability than a unified structure since only MSHR entries in a bank are searched upon a load miss. Each cache bank is locked up when MSHR entries of the bank are fully consumed. However, when load misses occur only in few cache banks due to access imbalance [TCT06], these cache banks can be easily locked-up by consuming all MSHR entries though other cache banks have ready MSHR entries.

To overcome the low utilization of banked MSHRs due to the access imbalance, Tuck et al. proposed a hierarchical cache miss handling architecture (HMHA) [TCT06]. However, HMHA was designed for L1 cache in CPUs. However, it is not suitable to GPGPUs. Many GPGPU applications have a regular streaming data access pattern which also have very high cache miss rate (e.g., more than 90%). GPGPUs are design to distributed these data accesses to each L2 bank (each memory channel). Thus, the imbalance accesses are rarely observed. Furthermore, HMHA's shared MSHR can be bottleneck in such high cache miss rates. The section 5.6 discusses HMHA in detail.

Based on this observation, we propose a tag shared MSHR array (TSMA) that is highly scalable to maximize GPGPUs' hardware multithreading. The TSMA shares the tags of cache blocks to remove fully-associative search in conventional MSHRs. This tag sharing, however, brings two side effects. First, if all the cache block's tags in a cache set are shared, then a cache miss in the cache set locks the entire cache until one of the shared tags is released. Second, sharing the cache tag evicts the corresponding cache block. This cache block eviction is earlier than a normal cache block eviction in conventional MSHRs that evict a cache block when it is filled. This early eviction can hurt the performance by decreasing the cache data locality degree. To relieve these problems, we also present a hybrid TSMA (HTSMA) consisting of a small size of a conventional MSHR and a large size of TSMA. HTSMA prevents the cache lock-up and relieves the early cache block eviction via the small conventional MSHR.

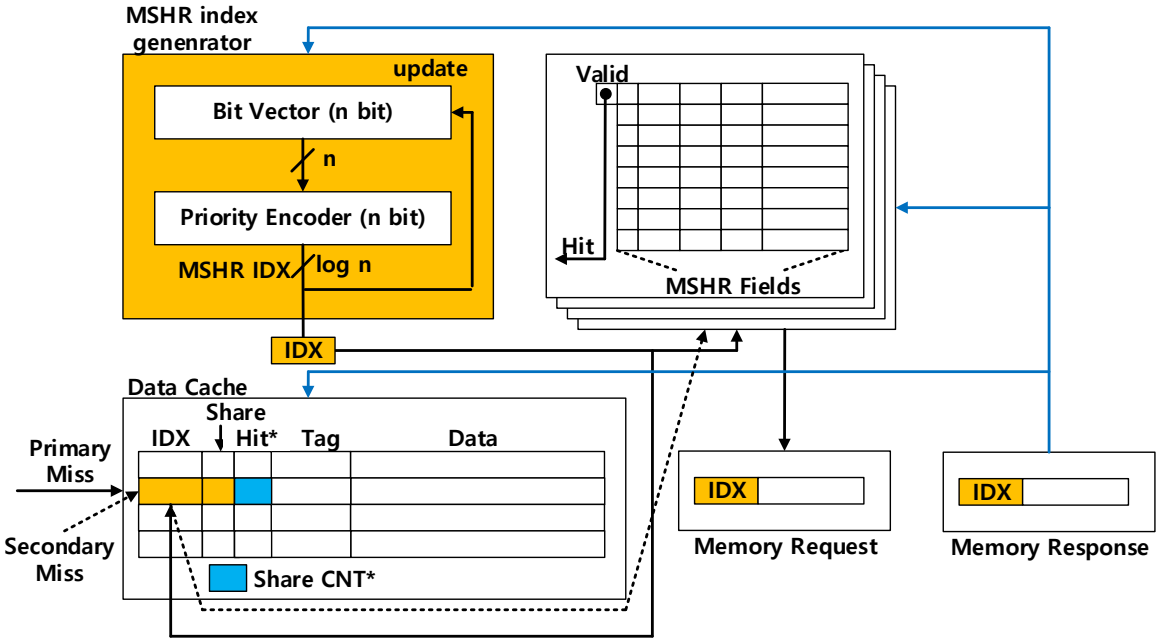

Figure 5.2: Tag Shared MSHR Array Structure with N MSHR entries: * are used in Hybrid Structure

5.1 Tag Shared MSHR Array

The tag shared MSHR array (TSMA) provides a high degree of scalability by sharing the tag and tag comparator in the cache. TSMA consists of the MSHR index generator, an IDX register, and a tagless MSHR array as shown in Figure 5.2. The IDX register provides the available MSHR index. An MSHR entry in the tagless MSHR array is the same as a conventional MSHR entry without a tag and tag comparator. TSMA also adds an MSHR array index (IDX) and a share-bit tag in each cache block.

Upon a cache miss (a primary miss), the MSHR array is accessed with the IDX register. Then, it sets the valid bit of target MSHR entry, fills the first field, and generates a memory request with the IDX to fetch the data from lower level memory. At the same time, a cache block is chosen by the replacement policy for tag sharing. Then, TSMA replaces the tag with the cache miss's tag, sets the share-bit to 1, and updates the IDX value. Upon a secondary

miss, it actually hits the cache but the share-bit is 1. A cache hit on the shared tag, which the share-bit value is 1, indicates that it is a secondary miss. Thus, a cache miss is always a primary miss in TSMA. TSMA accesses the MSHR array with the IDX value of the cache block, and fills an MSHR field in the target MSHR entry. When a memory response arrives, it accesses the MSHR array with IDX. Then, it releases the target MSHR entry, and fills the data in the corresponding (pending) registers and the target cache block.

The MSHR index generator generates an index of one of available MSHR entries and updates the IDX register as shown in Figure 5.2. The generator consists of a bit vector and a priority encoder. The length of the bit vector and the input bit width of the priority encoder are the same as the number of MSHR entries in the MSHR array. The size of the IDX register is log(the number of MSHR entries). All the bits in the bit vector are initially set to 1, which indicates the corresponding MSHR entry is available. Upon a primary miss, it reads the IDX register and sets the corresponding bit of the bit vector set to 0, which indicates the corresponding MSHR entry is consumed by the primary miss. When the processing of the primary miss is done in the cache and the MSHR array, the generator updates the IDX register value. When a memory response arrives, the corresponding bit in the bit vector respect to the IDX of the memory response is reset to 1 since the corresponding MSHR entry is released.

TSMA requires a simple modification on a cache and a memory request/response. The MSHR array index (IDX) and a share-bit are added to each cache block. The IDX tag is updated upon a primary miss. The share-bit sets to 1 upon a primary miss and resets to 0 when the cache block is filled. The IDX is also added to a memory request and a memory response. Here, we assume that the lower level memory maintains the IDX value. Since GPGPU has a two-level cache, two IDXs are stored in a memory request/response in our TSMA.

```
if shared_cnt==size(cache set)-1:
    access s-MSHR
else:
    if Hit_Tag==1:  // Try to use s-MSHR
        access s-MSHR
        if s-MSHR is full:
            access TSMA
    else:
        if Empty-bit==0:     // s-MSHR is not empty
            access s-MSHR    // it can be 2nd miss in s-MSHR
            if s-MSHR is missed:
                access TSMA      // Critical Path
        else:
            access TSMA
```

Listing 5.1: Pseudo Code for HTSMA

5.2 Hybrid Tag Shared MSHR Array

TSMA can provide a large number of MSHR entries without a high cost in terms of power and area due to the elimination of the fully-associative tag search. Still, TSMA has two side effects: the cache lock-up by sharing all the tag in a cache set and the early cache block eviction. To alleviate the side effects in TSMA, we propose a hybrid TSMA (HTSMA) consisting of TSMA with a small conventional MSHR (s-MSHR) that is from 1/8 to 1/4 size of TSMA. HTSMA adds three components to the TSMA: a hit-bit tag in each cache block, a share-bit counter in each cache set as shown in Figure 5.2, and an single empty-bit flag in the s-MSHR. A cache hit sets the hit-bit tag to 1, and a cache miss resets the hit-bit to 0. The share-bit counter counts the number of cache blocks those share-bit values are 1. The empty bit tag is set to 1 when the s-MSHR is empty.

HTSMA prevents the cache locked-up caused by sharing all the tags in a cache set by reserving one cache block for s-MSHR. The share-bit counter counts the number of shared cache blocks in a cache set. HTSMA does not allow the tag sharing and tries to use s-MSHR if the number of shared cache blocks reaches the cache set's number of blocks-1.

HTSMA also keeps track of hit cache blocks, then it tries to use the s-MSHR instead of

TSMA to reduce the early evictions if the hit cache block is chosen for tag sharing. Upon a cache miss, HTSMA checks the chosen cache block's hit-bit tag. If the tag is 1, then it tries to use normal cache miss handling with s-MSHR if s-MSHR is not fully consumed. Otherwise, it tries to use TSMA. Yet, unlike TSMA, the distinction of a cache miss type is unclear in the HTSMA because two MSHRs coexist. A cache miss in HTSMA can be one of the following cache miss types: a primary miss or a secondary miss in s-MSHR, or a primary miss in TSMA. If the chosen cache block's hit tag is 0 upon a cache miss, then HTSMA tries to use TSMA, but it checks s-MSHR first since the cache miss can be a secondary miss in s-MSHR if s-MSHR is not empty. Thus, if it misses s-MSHR, then it accesses TSMA as a primary miss. To reduce unnecessary accesses to s-MSHR, HTSMA checks the empty-bit flag before it compares the tags in s-MSHR since a cache miss is always a primary miss in TSMA if s-MSHR is empty. We believe that the empty-bit is worthwhile since there are many GPGPU applications with very little data locality. The HTSMA mechanism is explained in Listing 5.1 with a pseudo code.

5.3 Experiment Methodology

This section describes the GPGPU simulator and the benchmark suits used in the experiment. First, it explains the modification applied to the GPGPU-Sim simulator. Then, the classification rules for benchmark groups is described.

5.3.1 GPGPU Simulator

We extend GPGPU-Sim 3.2.1 [BYF09] to support TSMA and HTSMA. We also add the hash function for the cache set index [NBC14] in both the L1 and L2 data caches. The baseline configuration details are shown in Table 5.1. The number of MSHR entries in the baseline configuration is 32 in the L1 data cache and L2 bank. Each MSHR entry has 8 MSHR fields. This configuration of MSHRs is the same as the configuration of GTX480 [NVIc] in the GPGPU-Sim. We use 32K size 8-set associative cache for the L1 data cache and 200

Parameter	Value
# Compute Units	15
SIMD unit width	32
Warp Size	32
# Threads / Core	max 1024
# Registers / Core	32768
Shared Memory / Core	32KB (16 banks)
Constant Cache / Core	8KB (2-way, 64B line, LRU)
Texture Cache / Core	16KB (8-way, 128B line)
L1 Data Cache (128B line, LRU)	32KB (8-way)-default, 16KB (4-way) 32KB4W (4-way), 48KB (6-way), 64KB (8-way)
MSHR in L1D cache	32 entries (8 fields / entry)
Shared L2 cache	128KB / MC (8-way, 128B line, LRU),
MSHR in L2D cache	32 entries (8 fields / entry)
Core Clock	1400 Mhz
Interconnect Clock	1400 Mhz
Memory Clock	924 Mhz
Interconnection	crossbar, 32B channel width
# Memory Channels	6, FR-FCFS [RDK00]
DRAM min latency	100, 200(default), 400 cycles
DRAM request queue size	32
DRAM	16 DRAM banks / MC
GDDR5 Timing	tCL=12, tRP=12, tRC=40, tRAS=28, tRCD=12, tRRD=6
Memory Channel BW	8 (Bytes/Cycle)

Table 5.1: Simulator Configuration

cycles for the default minimum DRAM latency for the default configuration.

We use the *on fill* cache block allocation policy as a default policy. GPGPU-Sim's default allocation policy is *on miss*, which is unnatural. The *on miss* policy allocates (or reserves) a cache block on a cache miss, then fills the data later when the memory response arrives. The cache block allocation acts as a cache block eviction like TSMA. Another problem of *on miss* policy is that a cache set can be locked up when all the cache blocks in a cache set are allocated. However, the *on fill* policy maintains the cache block until it is filled, and it does not have the locked up cache set problem of the *on miss* policy.

5.3.2 Benchmarks

We examine 28 CUDA benchmarks from five different benchmark suites: CUDA SDK [NVI11a], GPGPU-Sim benchmark [BYF09], Rodinia [CBM09a], Parboil [SRS12], and SHOC [DMM10]. We divide these benchmarks into four groups based on MSHR sensitivity and data locality in L1 data cache or L2 unified cache: type-0(insensitive), type-I(little locality in L1 and L2), type-II(data locality in L2), and type-III(data locality in L1 and L2).

If a benchmark performance is not affected by the different number of MSHR entries, then it is considered type-0. The following 10 benchmarks are grouped in type-0: CUDA SDK (Histogram, MonteCalro), GPGPU-Sim benchmark (CP, MUM, STO), Rodinia (SRAD_ver1), Parboil (CUTCP, MRI-Q, Stencil), and SHOC (Sort).

The remaining 18 benchmarks are grouped based on a primary cache miss rate in the L1 and L2 data cache. If a benchmark's primary cache miss rates of both the L1 and L2 data caches are more than 90%, then it is grouped in type-I. If the primary miss rate is more than 90% only in the L1, then it is grouped in type-II. The remaining benchmarks are grouped in type-III. Table 5.2 provides the benchmark list and kernel information of these 18 MSHR sensitive benchmarks: the number of kernels(#K), the number of thread blocks (#CTAs), and the number of threads per thread block (T/CTA).

Type	Name	Abbr.	#K	CTAs	T/CTA
I	BlackScholes [NVI11a]	BLS	1	480	128
	Reduction [NVI11a]	RED	2	256	256
	ScalarProd [NVI11a]	SP	1	1024	256
	Triad [DMM10]	TR	1	32768	128
II	Convolution-Separable [NVI11a]	CS	2	1024	128
	FastWalsh-Transform [NVI11a]	FWT	3	1024	256
	MatrixMul [NVI11a]	MM	1	600	1024
	Scan [NVI11a]	SCAN	3	416	256
	Sorting-Networks [NVI11a]	SN	4	256	512
	LBM [SRS12]	LBM	1	18000	120
III	Transpose [NVI11a]	TP	8	4096	256
	LPS [BYF09]	LPS	1	2048	128
	WP [BYF09]	WP	1	72	64
	Breadth First Search [CBM09a]	BFS	2	256	512
	SRAD_ver2 [CBM09a]	SRAD2	2	4096	256
	StreamCluster [CBM09a]	SC	1	128	512
	SAD [SRS12]	SAD	3	128640	61
	SPMV [SRS12]	SPMV	1	765	192

Table 5.2: GPGPU Benchmarks Description

5.4 Experiment Result

This section is organized as follows. It analyzes the performance with varying MSHR size over the baseline configurations. It also evaluates the performance of our tag shared MSHR structures. Finally, it shows the sensitivity study results on different cache configurations and different DRAM latencies.

5.4.1 Optimal MSHR Size

The performance with varying size of conventional MSHR over the baseline (32 MSHR entries in L1 and L2 bank) is shown in Figure 5.3. Benchmarks' performances in type-0 are barely affected by different MSHRs' sizes. On the other hand, more MSHR entries usually increase the performance of the MSHR sensitive benchmarks (type-I, type-II, type-III). However, the unlimited size of MSHR entries in the L1 data cache (U, 32) over the baseline does not provide the performance improvement since 32 MSHR entries in the L2 bank are not sufficient to handle all the memory requests issued from the L1. Among the MSHR sensitive benchmarks, type-I shows the maximum performance improvement (43%) over the baseline since type-I has little data locality in both the L1 and L2 cache.

In this experiment, we find the optimal size of MSHR in the L1 data cache and L2 bank in terms of cost and performance improvement. Finding the optimal MSHR size, we increase L1 MSHR size with unlimited MSHR in L2 bank from (16, U) to (128, U). The performance increases up to (48, U), then it saturates with larger than 48 MSHR entries in the L1. With 48 MSHR entries in L1, we also increase L2 bank MSHR size from (48, 64) to (48, 160). The performance increases up to (48, 96), then it saturates with larger MSHR. Thus, we use (48, 96) MSHR as the optimal MSHR size in default DRAM latency configuration (200 cycles). In the same way, we find the optimal MSHR size in 100 and 400 cycles of the minimum DRAM latency configurations. They are (48, 64) and (64, 128), respectively.

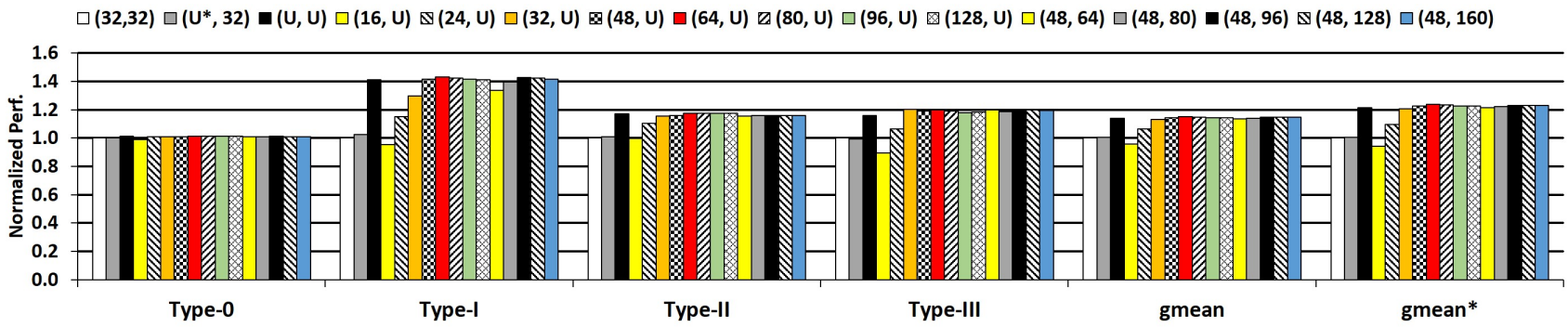

Figure 5.3: Performance with varying MSHR size: U*=Unlimited, gmean*=gmean except type-0

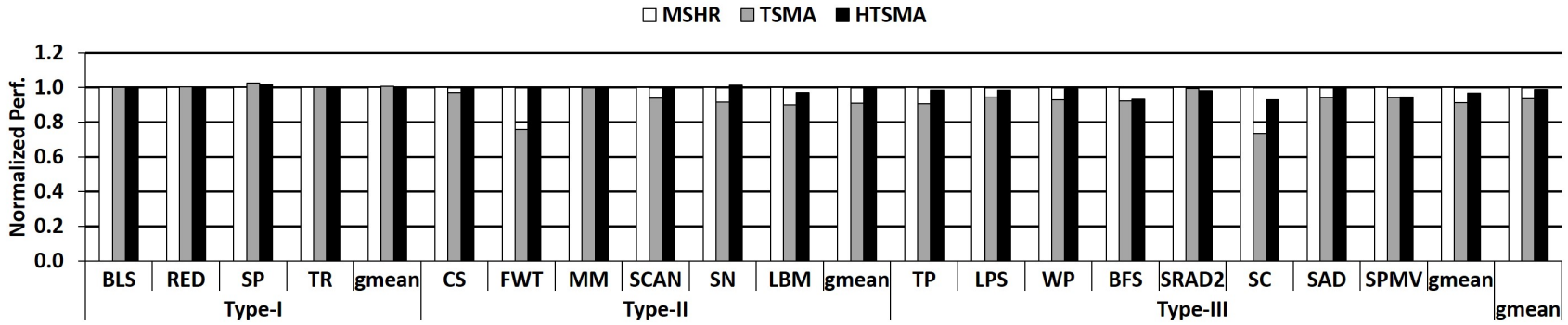

Figure 5.4: TSMA and HTSMA performance over MSHR in MSHR sensitive benchmarks

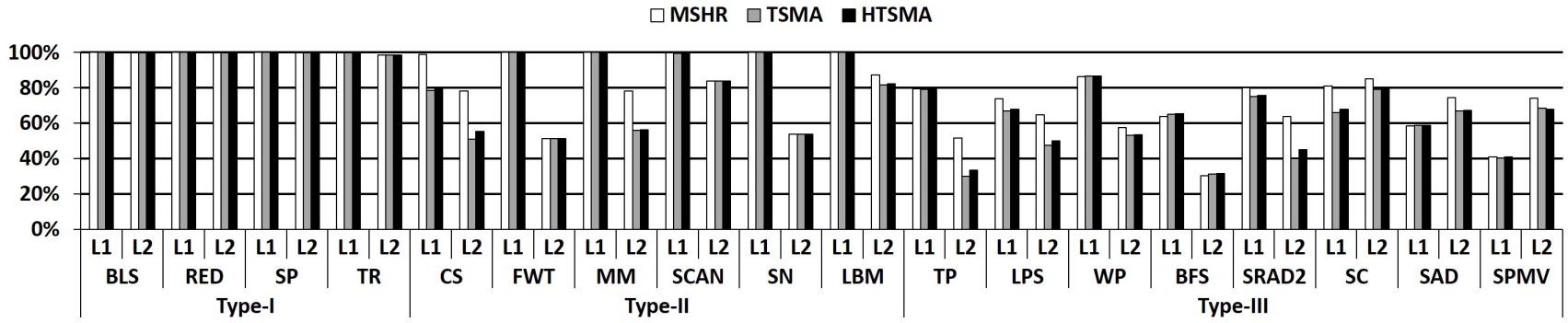

Figure 5.5: Cache miss rate of MSHR, TSMA, and HTSMA in MSHR sensitive benchmarks

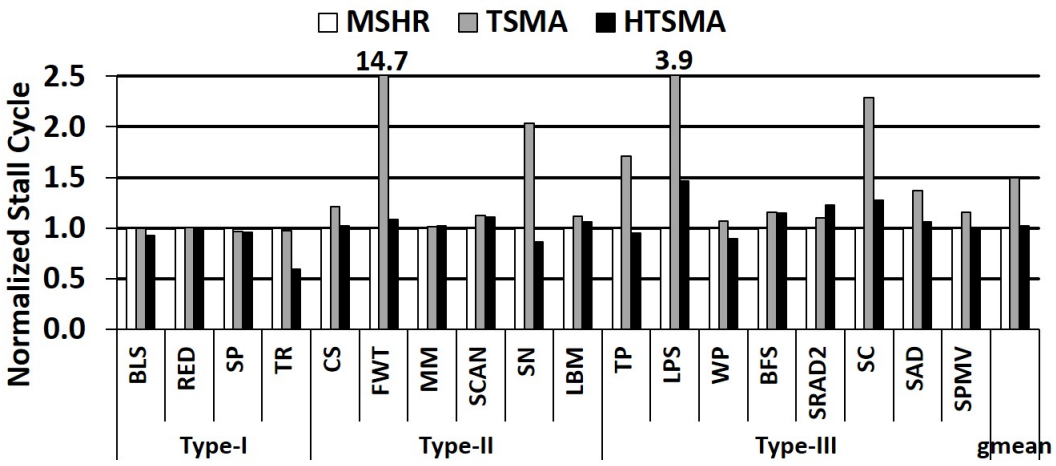

Figure 5.6: L1 data cache Stall Cycles normalized MSHR

5.4.2 Analysis Tag Shared MSHR

Figure 5.4 shows the performance of TSMA and HTSMA over the conventional MSHR in 18 MSHR sensitive benchmarks. In this default configuration, HTSMA adds the size of 4 and 8 conventional MSHR (s-MSHR) in the L1 and L2 TSMA, respectively. The size of s-MSHR in all HTSMA configuration is shown in Table 5.3. Overall, TSMA and HTSMA provide 93.4% and 98.5% performance of MSHR, respectively. In type-I, there are no noticeable differences between the three MSHR configurations. However, only TSMA's performance is decreased by 9.1% in type-II. In type-III, TSMA and HTSMA show 8.9% and 3.4% performance decrease over the conventional MSHR, respectively.

MSHR vs TSMA: TSMA's performance decrease against a conventional MSHR (MSHR) in type-II and type-III can come from the two side effects: cache lock-ups by fully shared tags and early cache block evictions. Interestingly, the effect of cache lock-ups by fully shared tags is much more significant than the effect of early cache block evictions. Figure 5.5 and 5.6 provides the explanation. The miss rate of TSMA is equal or less than the miss rate of MSHR in most of type-II and type-III benchmarks in Figure 5.5. The increase of miss rate by early block evictions in TSMA is observed only in BFS. Especially, the miss rates are almost the same in FWT, SCAN, and SN while TSMA's provide only 75.7%, 93.7%, and 91.6% of the conventional MSHR performance. The only explanation for TSMA's decreased performance

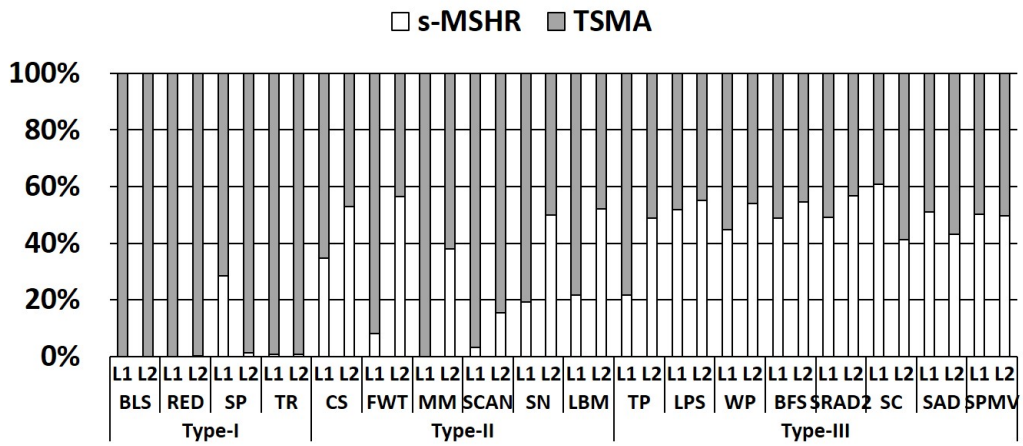

Figure 5.7: Access Ratio of s-MSHR and TSMA in HTSMA

is the cache lock-up. Figure 5.6 presents the three MSHR's L1 data cache's stall cycles by load accesses normalized to MSHR. TSMA's stall cycles are significantly increased in FWT, LPS, and SC over the conventional MSHR mainly due to the cache lock-ups by sharing all the tags. Overall, TSMA increases the cache stall cycles by 50% over the conventional MSHR.

On the other hand, the cache lock-up can result in the increase of data locality since cache blocks can not be evicted while the cache is locked up. TSMA's cache miss rate(L1 or L2) is lower than conventional MSHR in CS, LBM, TP, LPS, WP, SRAD2, and SPMV. However, TSMA's performance is still lower than the performance of the conventional MSHR in those benchmarks because the performance gain by increased cache data locality is much lower than the performance loss by the cache lock-ups.

TSMA vs HTSMA: HTSMA recovers the most performance drops caused by TSMA in type-II and type-III except for BFS and SPMV. This performance improvement of HTSMA over TSMA primarily comes from avoiding the cache lock-ups. Figure 5.6 shows that HTSMA removes most of the extra cache stall cycles from TSMA in the L1 MSHR. Overall, HTSMA's stall cycles are almost same as s-MSHR. Avoiding the cache lock-up can result in an increase in the cache miss rate. Thus, HTSMA's cache miss rate is equal to or higher than the miss rate of TSMA. In BFS and SPMV, HTSMA increases performance by 1% over TSMA.

Cache Config	TSMA size (48, 64)	(48,96)	(64,128)
32K	(4,8)	(4,8)	(8,16)
16K	(12,8)	(12,8)	(12,16)
32K4W	(8,8)	(8,8)	(12,16)
48K	(8,8)	(8,8)	(8,16)
64K	(4,8)	(4,8)	(8,16)

Table 5.3: s-MSHR configuration in HTSMA

The small conventional MSHR (s-MSHR) in HTSMA is the source of the performance gain over TSMA. The access ratios of both MSHRs in HTSMA are shown in Figure 5.7. Only SP's L1 HTSMA shows a high s-MSHR access ratio in type-I. This is primarily caused by avoiding the cache lock-up since the type-I miss rate is close to 100%. HTSMA's s-MSHR access ratio varies in type-II. However, there is a pattern that s-MSHR ratio in L2 HTSMA is usually larger than the ratio in the L1 HTSMA in type-II. Since the L2 cache miss rate is high in type-II, most s-MSHR accesses in the L2 HTSMA are to avoid the cache lock-ups. In type-III, all benchmark's s-MSHR access ratios in both L1 and L2 are more than 40% because of the type-III higher data locality than type-I and type-II.

5.4.3 Sensitivity Study

Cache size and associativity can affect the performance of TSMA and HTSMA. The data locality in a small size of the cache can be easily lost by a few early cache block evictions, and a lower degree associative cache can be locked up more frequently than a higher degree associative one. Figure 5.8 shows the performance of the three MSHRs with all five L1 data cache configurations shown in Table 5.1.

There are no noticeable performance differences in type-0 and type-I except for TSMA's performance with 16K in type-I. It shows a 5.9% performance drop mainly caused by the cache lock-up. In type-II, TSMA shows a 9% performance drop over the conventional MSHR

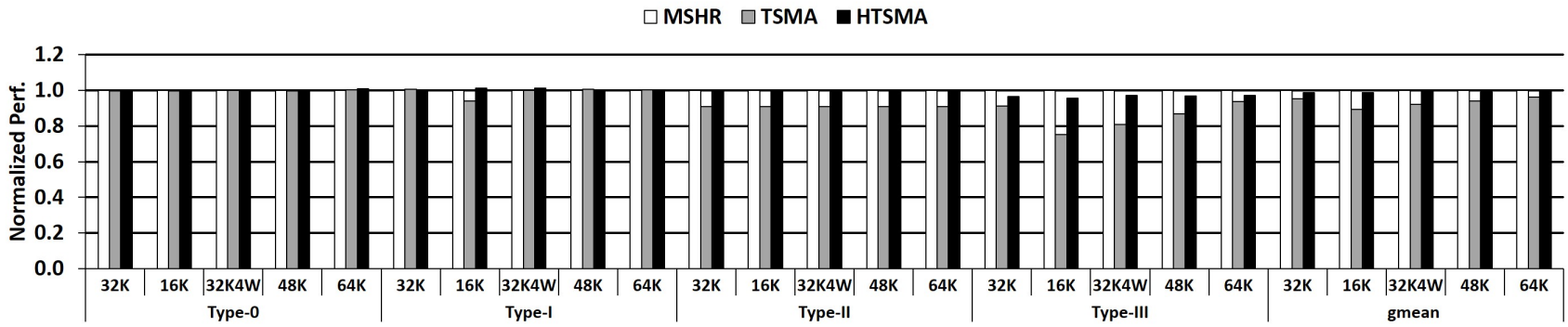

Figure 5.8: MSHRs Performance with varying Cache Configuration

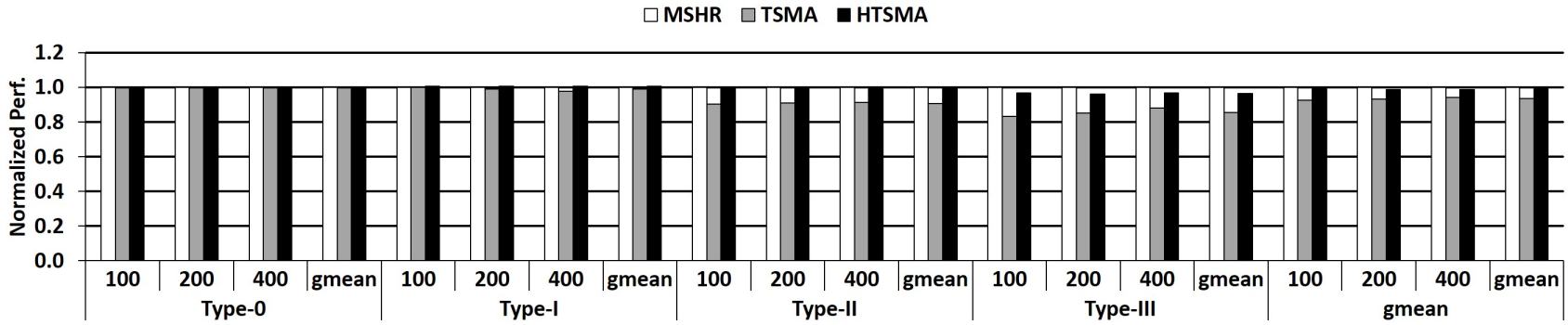

Figure 5.9: MSHRs Performance with varying DRAM Minimum Latency

while HTSMA recovers most of the performance drop in all the cache configurations. TSMA's performance decrease varies from 24.8% to 5.9% depending on the L1 cache configuration in type-III. The smaller size and lower degree associative decreases the performance more significantly. However, HTSMA recovers most of the performance drop from 95.7% to 97.2% of the conventional MSHR's performance in type-III.

Overall, the performance of TSMA varies from 89.4% to 96.4% of the conventional MSHR's performance, depending on the cache configurations. Unlike TSMA, HTSMA provides uniform performance that is from 98.8% to 99.4% of the conventional MSHR performance in all the cache configurations. However, the size of s-MSHR used in HTSMA is different depending on the cache configuration as shown in Table 5.3 since the two side effects of TSMA becomes worse as the cache size or the associativity decreases.

Figure 5.9 shows the performance of TSMA and HTSMA over the conventional MSHR with varying DRAM latency. We varied the DRAM latency by changing the minimum DRAM latency in the simulator. Each result in Figure 5.9 is the geometric mean of results in all five cache configurations. The performance of TSMA and HTSMA are not significantly affected by varying DRAM latency though the number of MSHR used in three MSHRs increases as the latency increases. Overall, TSMA and HTSMA shows 93.5% and 99.0% of the conventional MSHR's performance, respectively. The sizes of the conventional MSHR and TSMA are (48, 64), (48, 96), and (64, 128) in 100, 200, and 400 cycles, respectively. The sizes of TSMA used in HTSMA are the same as TSMA, and the sizes of s-MSHR used in HTSMA are shown in Table 5.3. The s-MSHR size in HTSMA is the optimal size considering the extra cost and performance gain by increasing the size.

5.4.4 Comparison to Set Associative MSHR

Set associative MSHRs are another way to increase the scalability of MSHRs. The access time, power consumption, and area cost usually decreases as a MSHR's set associativity decreases. However, a set associative MSHR cannot handle a cache miss when the corre-

Cache Config		Associativity			
		1xASSOC	**2xASSOC**	**4xASSOC**	**8xASSOC**
L1 (64 MSHR)	32K	8-way	16-way	32-way	Fully-assoc
	16K	4-way	8-way	16-way	32-way
	32K4W	4-way	8-way	16-way	32-way
	64K	8-way	16-way	32-way	Fully-assoc
L2 (128 MSHR)	128K	8-way	16-way	32-way	64-way

Table 5.4: Set Associative MSHR configuration

sponding set is fully consumed. Therefore, GPGPU's performance can be decreases as a MSHR's set associativity decreases due to imbalance accesses among MSHR's sets. TSMA can be also considered as a set associative MSHR because the cache's associativity can limit the utilization of TSMA. When all the tags in a cache set are used for TSMA, TSMA can not handle further cache misses in the cache set although it has available entries. Thus, the cache's set associativity can be considered as TSMA's set associativity in the worst case. To measure the provided set associativity of TSMA (HTSMA), we compare TSMA (HTSMA) with different set associative MSHRs.

To properly configure a set associative MSHR with regard to the cache's associativity for TSMA, the 400 cycle's minimum dram latency configuration which has 64 and 128 MSHR entries in L1 and L2 respectively is used for the comparison. The MSHR's associativity starts at the associativity of the cache and increases up to eight times of the cache associativity. The L1 with 48K configuration is since its associativity is six. Table 5.4 shows set associative MSHR configurations. The configuration of both L1 and L2 MSHR's set associativity starts from the corresponding cache's set associativity (1xASSOC) up to 8 times higher set associativity (8xASSOC) as shown Table 5.4.

The performances of various set associative MSHRs, TSMA, and HTSMA against a conventional MSHR are shown in Figure 5.10: (a) based on L1 cache configuration and (b) based on the benchmark group. The performance decreases as the MSHR's set associativity

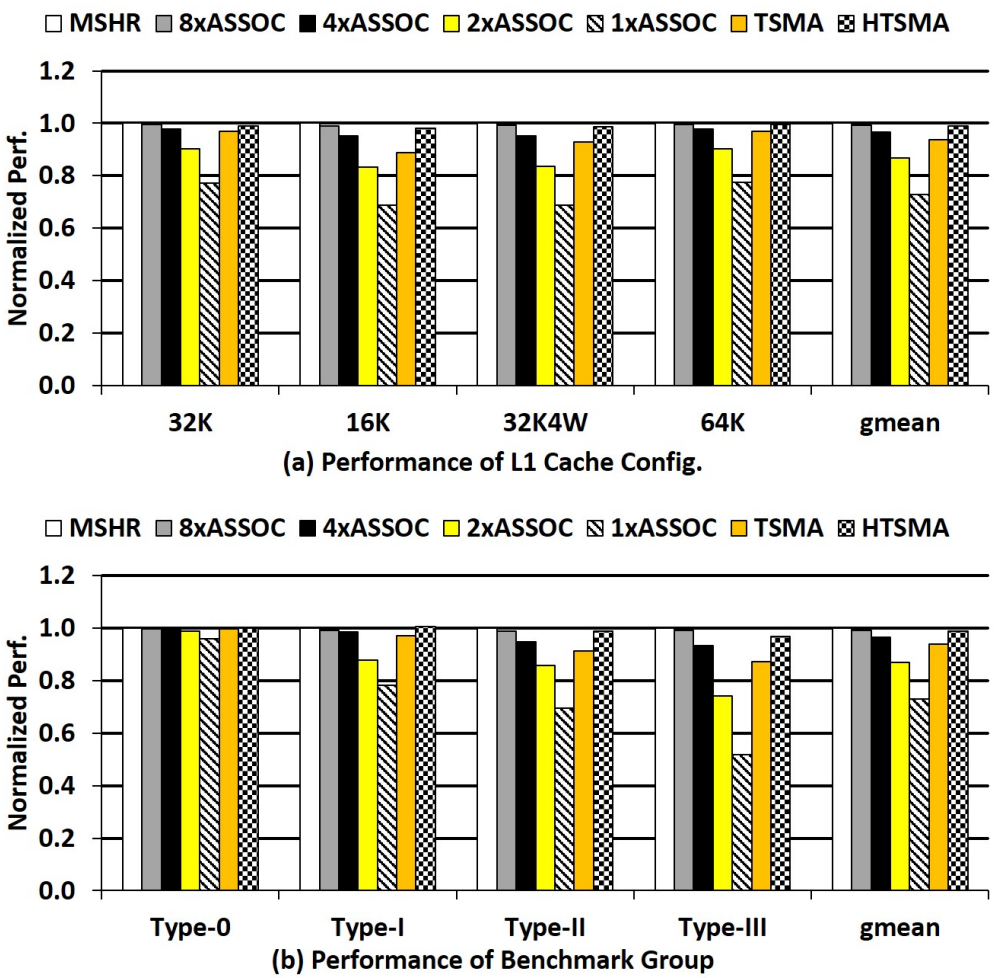

Figure 5.10: Various MSHR Performance normalized to conventional MSHR's performance

decreases. Overall, the set associative MSHRs from 8xASSOC to 1xASSOC configurations reach 99%, 97%, 87%, and 73% of the conventional MSHR's performance respectively. There is no noticeable performance drop in 8xASSOC MSHRs since since 8xASSOC is the highest set associative MSHR and the fullly associative MSHR in some L1 configurations(32K and 64K). However, the performances are significantly dropped from 2xASSOC to 1xASSOC. Set associative MSHRs with 16K and 32K4W L1 configurations result in more performance drop than 32K and 64K L1 configurations as shown in Figure 5.10(a) since 16K and 32K4W L1 MSHR's associativities are half of 32K and 64K L1 MSHR's associativities that causes more imbalance accesses among MSHR sets.

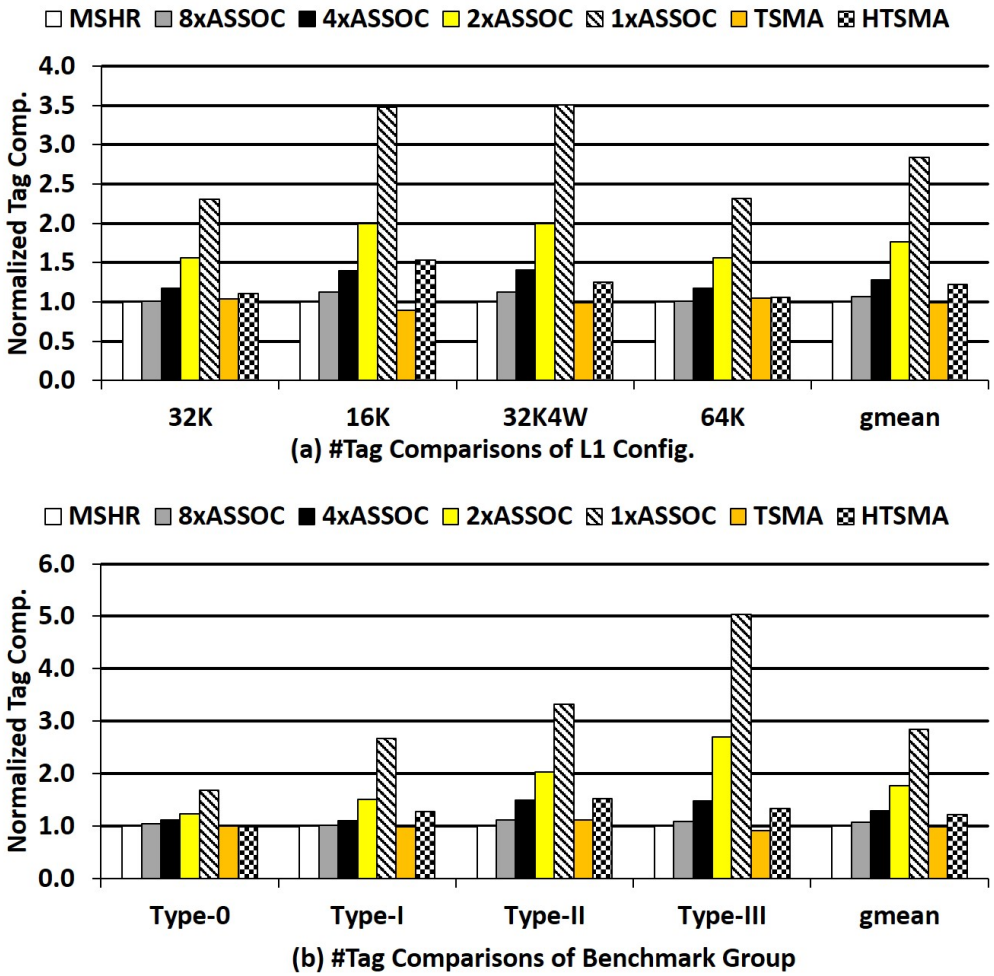

Figure 5.11: Various MSHRs' tag comparisons normalized to conventional MSHR's tag comparisons

TSMA's performance can be same as 1xASSOC in the worst case since its associativity limits by the cache's associativity. However, TSMA performance (94% of the conventional MSHR) is higher than all the 2xASSOC configurations, and it is close to 4xASSOC's performance. We believe that this TSMA high performance comes from the fact that the number of cache blocks is much higher than the number of MSHR entries. For example, the 32K L1 has 128 cache blocks and 32 cache sets with 8-way set associativity. All the cache accesses (misses) are distributed over 32 cache sets. Still, 64 MSHR entries have 8 sets with 8-way associativity in 1xASSOC configuration. All the cache misses are distributed over 8 sets. Thus, the imbalance accesses among sets in TSMA can be much less than 1xASSOC MSHR.

Figure 5.10(b) shows the performance of the various MSHRs in different benchmark suits. Type-0 is barely affected by decreasing MSHR's set associativity. However, the performance is significantly decreases in other types. The performance drops from type-I to type-III reveal the relationship between the cache locality and imbalance accesses of set associative MSHRs. Among type-I, type-II, and type-III, type-I has the least cache locality (miss rate >90% in L1 and L2), type-II has the medium cache locality (miss rate >90% only in L1), and type-III has the most cache locality (miss rate $\leq$ 90% in L1 and L2). However, the performance order of these three types is inverse to the cache locality: type-I >type-II >type-III as shown in Figure 5.10(b). Applications with more cache locality can increase the imbalance accesses among MSHR sets since their memory accesses have a repeated access pattern.

A cache miss that accesses to the MSHR's fully consumed set is not processed, but it performs the tag comparison in the MSHR's set. Thus, the degree of imbalance accesses in set associative MSHRs can be indirectly measured by counting the number of tag comparisons in MSHR sets. Figure 5.11 shows the number of various MSHRs' tag comparisons normalized to the number of convention MSHR's tag comparisons: (a) based on L1 cache configuration and (b) based on the benchmark group. The result is a sum of L1 and L2 MSHR's tag comparisons. The number of tag comparisons significantly increases as the MSHR's set associativity decreases. These significantly increased tag comparisons in set associative MSHRs have a negative impact on the performance that matches the performance

results shown in Figure 5.10. Similarly, benchmarks with higher cache locality result in more tag comparisons in set associative MSHRs as shown in Figure 5.11(b). Note that the number of compared tags in the set associative MSHR's tag comparison is much less than one in the conventional MSHR's tag comparison since set associative MSHRs compares tags only in the MSHR set, not the entire tags of MSHR.

5.5 Power, Area, and Access Time

We use CACTI v4.0 tool [Dav06] to estimate the power consumption, area, and access time of conventional MSHR, set associative MSHR, TSMA, and HTSMA with $130nm$ technology. The estimation of a s-MSHR with four entries in HTSMA is not shown since it is too small to run the CACTI tool. (Recent CACTI v5.3 that supports up to $32nm$ does not work properly with our MSHR configurations due to MSHR's small size.) We consider an MSHR entry with 8 MSHR fields as a 56-byte cache block for CACTI simulation. We also use full associativity and 1-set associativity (direct-map) in conventional MSHR and TSMA, respectively. Unlike a cache, TSMA does not have tags and tag comparators. However, we did not exclude them to compensate for the extra components' costs in TSMA or HTSMA: share-bit tag (share-bit counter), hit-bit tag, and IDX tag in each cache block, MSHR index generator, and IDX register.

We calculate the extra cost of TSMA and HTSMA based on the following assumptions: 32-bit tag and tag comparator, 599 transistors for a 64-bit priority encoder in $130nm$ [AMI15], 1917 transistors for a 32-bit comparator in $150nm$[AGP13], and 6 transistors for a 1-bit SRAM cell. Our calculation shows that the total extra cost is roughly from 29% to 41% of TSMA's tag and comparator cost in CACTI. Thus, we believe that our estimation of TSMA and HTSMA in CACTI is slightly overestimated.

MSHR	**Dyn**(mW)	**Leak**(mW)	**Area**(mm^2)
(48, 64)	6448.95	1078.31	10.01
(48, 96)	7107.34	1264.31	10.66
(64,128)	8590.49	1682.81	11.95
TSMA	**Dyn**(mW)	**Leak**(mW)	**Area**(mm^2)
(48, 64)	805.93	148.00	3.31
(48, 96)	867.44	168.93	3.49
(64,128)	922.49	220.10	4.63

HTSMA				
TSMA	s-MSHR	**Dyn**(mW)	**Leak**(mW)	**Area**(mm^2)
(48, 64)	(8, 8)	2207.92	319.51	5.81
	(12,8)	2451.88	377.63	6.02
(48, 96)	(8, 8)	2269.43	340.44	6.00
	(12,8)	2513.39	398.57	6.20
(64,128)	(8,16)	2517.49	438.11	7.29
	(12,16)	2761.45	496.24	7.49

Table 5.5: Power and Area Estimation

5.5.1 MSHR, TSMA, and HTSMA

The power and area estimations of MSHR, TSMA, and HTSMA are shown in Table 5.5. The result is the sum of all 15 SM's L1 MSHRs and 6 L2 banks' MSHRs. Removing full associativity decreases the power consumption and the area costs significantly as shown in the MSHR and TSMA table 5.5. Overall, TSMA's power consumption (dynamic+leakage) and area costs are only 12%, and 35% of conventional MSHR's power consumption and area costs, respectively. HTSMA's power consumption and area costs are roughly 33% and 59% of conventional MSHR's ones, respectively. Overall, the s-MSHR in HTSMA consumes about 63% and 41% of the entire power and the area in HTSMA, respectively. This result reveals the fully-associative structure's expensive cost in terms of power and area.

Table 5.6 shows that the access time of the conventional MSHR, TSMA, and HTSMA (s-MSHR). TSMA's access time is roughly half (from 50% to 53%) of MSHR's access time. HTSMA has three different access times: TSMA access, s-MSHR access, s-MSHR tag comparison and TSMA access (the critical path). The access time of HTSMA's critical path, however, is increased by 9% and 5% over MSHR's access time in the L1 and L2, respectively. This increased access time comes from the fact that the HTSMA's critical path consists of a s-MSHR miss then TSMA access that includes the s-MSHR tag comparison time and the access time of TSMA. On the other hand, the increased access time of HTSMA can be relieved by the 2-stage pipeline structure. This 2-stage pipeline structure performs (or skips) s-MSHR's tag comparison in the first stage, and it access s-MSHR's entry or TSMA in the second stage. In the pipelined HTSMA, the access time will be the tag comparison time of s-MSHR because the tag comparison time of s-MSHR is slightly larger than the access time of TSMA.

5.5.2 Set Associative MSHR

Figure 5.12 shows the relative power (dynamic and leakage), area, and access time of set associative MSHRs, TSMA, and HTSMA against the fully-associative conventional MSHR (MSHR) in the 400-cycle DRAM configuration: 64 and 128 MSHR entries in L1 and L2 respectively. Only few set associative MSHR configurations are estimated in the CACTI tool: 4-way and 8-way set associativity in L1 MSHR, and 8-way set associativity in L2 MSHR. The tool does not estimate properly in higher set associative configurations. The estimation of 4-way set associative MSHR in L1 corresponds to 1xASSOC configuration of 16K, 32K4W. The 8-way set associative MSHR estimation in L1 includes 2xASSOC configuration of 16K, 32K4W and 1xASSOC configuration of 32K, 64K as shown in Table 5.4.

Set associative MSHRs take less power, area, and access time than ones of the conventional MSHR as show in Figure 5.12. We expect all the values will be increased as the set associativity increases further. However, TSMA reduces all of them more than 1xASSOC MSHR while TSMA provides much higher performance than the performane of 2xASSOC

	MSHR, TSMA	
(L1, L2)	**L1**(ns)	**L2**(ns)
(48, 64)	1.50, 0.79	1.54, 0.80
(48, 96)	1.50, 0.79	1.62, 0.81
(64,128)	1.54, 0.80	1.69, 0.85

s-MSHR	**Access, Tag***	
(L1, L2)	**L1**(ns)	**L2**(ns)
(8, 8)	1.43, 0.85	1.43, 0.85
(12,8)	1.46, 0.88	1.43, 0.85
(8,16)	1.43, 0.85	1.50, 0.91
(12,16)	1.46, 0.88	1.50, 0.91

HTSMA			
TSMA	s-MSHR	**TSMA, s-MSHR, Tag*+TSMA**	
(L1,L2)	(L1,L2)	**L1**(ns)	**L2**(ns)
(48, 64)	(8, 8)	0.79, 1.43, 1.64	0.80, 1.43, 1.65
	(12,8)	0.79, 1.46, 1.67	0.80, 1.43, 1.65
(48, 96)	(8, 8)	0.79, 1.43, 1.64	0.81, 1.43, 1.66
	(12,8)	0.79, 1.46, 1.67	0.81, 1.43, 1.66
(64,128)	(8,16)	0.80, 1.43, 1.65	0.85, 1.50, 1.76
	(12,16)	0.80, 1.46, 1.68	0.85, 1.50, 1.76

Table 5.6: Access Time Estimation: Tag* is the tag comparison time of s-MSHR

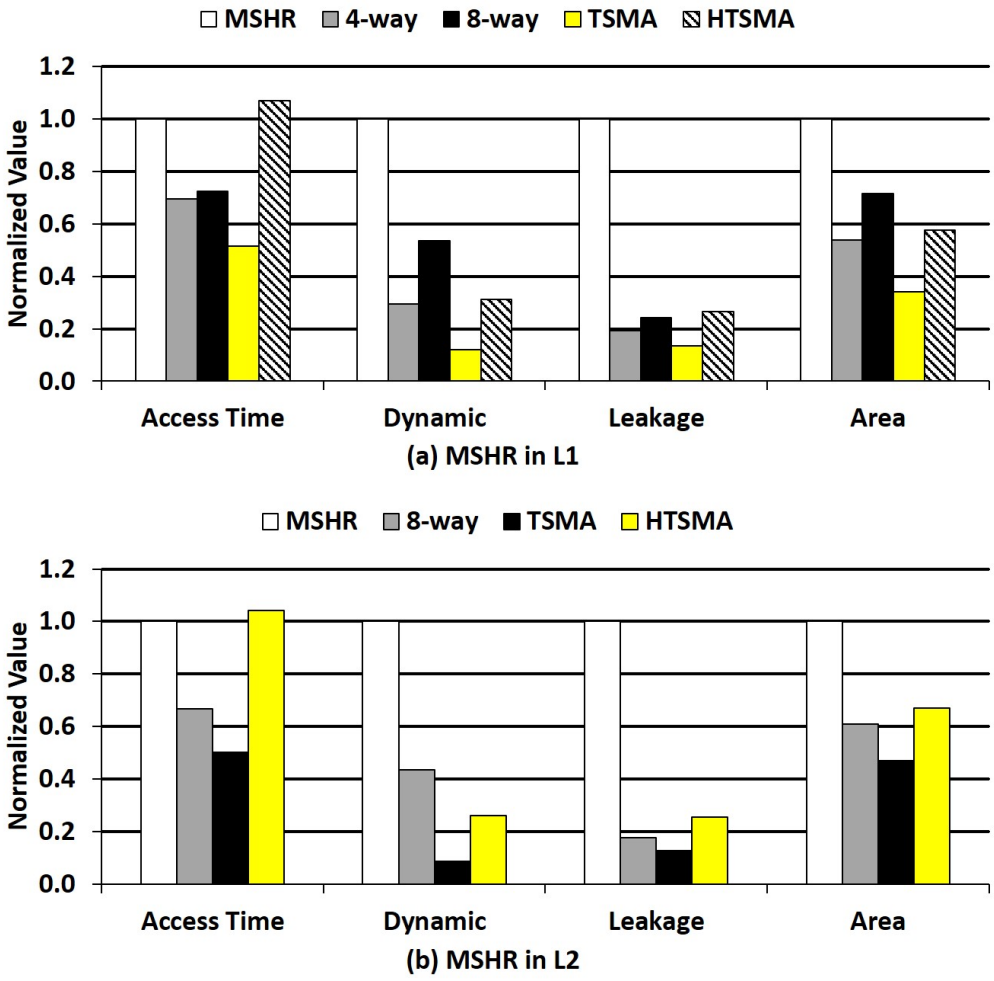

Figure 5.12: Power, Area, and Access time of Set Associative MSHRs

MSHR as shown in Figure 5.10. HTSMA consumes more power and area than TSMA, and HTSMA's access time is slightly higher than the conventional MSHR due to the addiction of s-MSHR. Still, HTSMA power and area consumption is close to 1xASSOC set associative MSHR configuration.

5.6 Related Work

Kroft [Kro98] introduced a Miss Status Holding Register (MSHR) to support non-blocking loads. An MSHR supports the non-blocking loads by keeping the information of a memory request: tag of the target cache block, the target register number, the format of a load

instruction, and other information. Scheurich and Dubois [SD88] proposed three MSHR schemes in multiprocessors and compared their tradeoffs. Sohi and Franklin [SF91] described the requirement of MSHR implementation in detail. They also evaluated the bandwidth advantages of banked MSHRs over unified MSHRs.

Farkas and Jouppi [FJ94] analyzed various MSHR implementations in term of the performance and implementation costs. They also mentioned, "In-Cache MSHR Storage" that replaces the cache tag and data block with an MSHR's tag and entry on a cache miss. It adds a "transit" bit tag in each cache block that indicates that the cache block is used as an MSHR entry. Thus, the in-cache MSHR can provide a large number of MSHR entries that is up to the number of cache blocks. However, it has the following critical problems. First, it adds many cycles' delay to the cache fill-time and the pending register fill-time. When fetch data arrives, the MSHR data must be read first before filling the cache block. Also, pending registers have to wait during this MSHR reading time. We believe that this delay is critical to its performance. Second, in-cache MSHR requires extra bit-wise processing to extract each MSHR data such as warp ID, register ID, and format information, etc., since MSHR data usually takes few bits. This extra processing adds another delay to fill pending registers. Furthermore, the in-cache MSHR has the cache lock-up and the early cache block eviction problems similar to TSMA.

Tuck et al. proposed the hierarchical cache a miss handling architecture (HMHA) [TCT06] that consists of two-level MSHR structure to improve the scalability of MSHR in CPUs. The two-level MSHR has a dedicated MSHR in each bank and a shared MSHR by all banks. When a dedicated MSHR is full and a cache miss occurs in the corresponding cache bank, one of the MSHR entries is moved to the shared MSHR. Thus, it can reduce the frequency of cache bank lock-ups due to access imbalance. The goal of HMHA is to increase banked MSHR's utilization by reducing the imbalance accesses among banks. However, HMHA is not suitable for GPGPUs due to the following reasons. Firstly, HMHA's shared MSHR can be the bottleneck in GPGPUs to support GPGPU's high degree of memory level parallelism. Many GPGPU applications have very little or no data locality. In such applications, the GPGPU

frequently consumes all MSHR entries in each bank due to a high cache miss rate. When most MSHR entries are consumed by a high cache miss rate, the shared MSHR in HMHA can be the bandwidth bottleneck since multiple banked MSHRs try to spill their MSHR entries to the shared MSHR. Secondly, GPGPU's L2 is already highly banked, but each L2 Bank requires a large number of MSHR entries. Lastly, the imbalance accesses between MSHR banks are not very common in GPGPUs' L2 banks since GPGPUs' data access patterns are commonly regular and GPGPUs distribute these accesses well among L2 banks or memory channels. Thus, more aggressive banking can be applied, but it will increase the cost of the interconnection network significantly.

C. Nugteren et al. claimed that 64 MSHR entries are used in NVIDIA's Fermi GPGPU core (MSHR in L1) based on their experiments [NBC14]. Increasing the number of MSHR entries increased only two benchmarks out of six in their experiment. However, we find that the number of MSHR entries in L2 has a more significant impact on the performance than the number of MSHR entries in L1. Our study also shows that the performance improvement can be limited by simply increasing the number of MSHR entries only in L1 since the small size MSHR in L2 can be the performance bottleneck.

5.7 Conclusion

GPGPUs require larger numbers of MSHR entries than CPUs to maximize the effectiveness of GPGPU's hardware multithreading. This work identifies the optimal number of L1-MSHR and L2-MSHR in GPGPUs. However, the optimal MSHR size is not feasible in conventional MSHRs. Set associative MSHR can be alternative but it hurts the performance significantly due to the imbalance accesses among MSHR's sets. Based on this observation, we propose a TSMA that can provide a sufficient number of MSHR entries for GPGPUs. By eliminating tags and tag comparators in MSHRs, TSMA easily increases the number of MSHR entries at a smaller cost than the cost of set associative MSHR while providing much higher performance than the performance of the corresponding set associative MSHR (1xASSOC). To alleviate

the side effects of TSMA, we also propose the hybrid TSMA that includes a small size of s-MSHRs. HTSMA provides 99% of conventional MSHR's performance only with 33% and 59% of conventional MSHR's power and area cost which are close to set associative MSHR's power and area cost.

CHAPTER 6

Conclusion

This dissertation presents three mechanisms to improve the effectiveness of hardware multithreading which leads to increase the performance in general-purpose graphics processing units (GPGPU): an adaptive warp scheduling policy, a warp throttling mechanism, and scalable cache miss handling architecture.

Firstly, it proposes an adaptive warp scheduling based on the evaluation of various warp scheduling policies' performance in different types of GPGPU applications. GPGPU applications are classified according to the data locality characteristics, and various warp scheduling policies are evaluated by varying the fairness. This work explains how unfair and fair policies effectively capture intra-warp locality and inter-warp locality respectively. Unfair policies can provide higher degree of latency hiding, and they are favorable to capture the intra-warp data locality. On the other hand, fair policies can capture more inter-warp locality. This results in the fact that unfair policies provide better performance than fair policies in applications with little data locality and high intra-warp data locality, but fair policies show higher performance than unfair policies in applications with high inter-warp data locality. Based on this observation, it devises locality-aware warp scheduling (LAWS) to exploit both unfair and fair scheduling policies. LAWS adaptively chooses the scheduling policy by measuring the data locality characteristics in the L1 data cache. Overall, LAWS improves the performance by 18% over the GTO scheduling policy in 16 scheduling sensitive benchmarks.

Secondly, this work proposes a simple warp throttling mechanism to improve the performance by reducing the intra-warp locality loss in the L1 data cache (L1D). Since many warps share the small size L1D in each streaming multiprocessor, each warp's data locality can be

easily interfered by other warps' cache accesses. Based on this observation, this study devises a throttling mechanism with little cost to improve the performance of GPGPUs further by reducing the intra-warp locality loss. In this work coupling with the LAWS, we achieve both locality-warp adaptive scheduling and throttling mechanisms with the minimum cost by sharing the functionality of the locality detection unit and exploiting the existing MSHR resources. This LAWS with throttling improves the performance 58% over GTO scheduling policy in 16 scheduling sensitive benchmarks.

Thirdly, it studies hardware prefetching mechanism on GPGPUs. It explains the challenges and solutions to apply the conventional hardware prefetching on GPGPUs, and it proposes an efficient hardware prefetching mechanism based on PC-based stride prefetcher exploiting the property of GPGPU's SIMT programming model. The stride detection across thread blocks used in ITPC offers new prefetching opportunities in GPGPU systems. Thread-block basis prefetch throttling mechanism presents the fine-grain control of prefetching throttling in GPGPUs.

Finally, it presents scalable cache miss handling architecture that can provide enough number of miss status holding registers (MSHR) to improve the effectiveness of hardware multithreading. Hardware multithreading on a cache miss has to be stalled when all the MSHRs are fully consumed. However, the scalability of conventional MSHRs is limited in terms of power and area due to the fully-associative structure. Based on this observation, we propose a tag shared cache miss handling architecture (TSMA) that can provide a sufficient number of MSHRs for GPGPUs. By eliminating tags and tag comparators in MSHRs, TSMA easily increases the number of MSHRs at a small cost. However, TSMA has two side effects that can decrease the performance. To alleviate the side effects of TSMA, we also propose the hybrid TSMA that includes a small size of conventional MSHRs. HTSMA provides 99% of conventional MSHR's performance only with 33% and 59% of conventional MSHR's power and area cost.

References

[AF] Tor M. Aamodt and Wilson W.L. Fung. "GPGPU-Sim 3.x Manual." http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual.

[AGP13] S. Abdel-Hafeez, A. Gordon-Ross, and B. Parhami. "Scalable Digital CMOS Comparator Using a Parallel Prefix Tree." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **21**(11):1989–1998, Nov 2013.

[AMD12] AMD. "Hegerogeneous Computing: OpenCL and the ATI RAdeon HD 5870 (Evergreen) Architecture." http://developer.amd.com/wordpress/media/2012/10/Heterogeneous_Computing_OpenCL_and_the_ATI_Radeon_HD_5870_Architecture_201003.pdf, 2012.

[AMI15] K. M. Ali, H. Mostafa, and T. Ismail. "High performance layout-friendly 64-bit priority encoder utilizing parallel priority look-ahead." In *Energy Aware Computing Systems Applications (ICEAC), 2015 International Conference on*, pp. 1–4, March 2015.

[BBH] Darrell Boggs, Aravindh Baktha, Jason Hawkins, Deborah T. Marr, J. Alan Miller, Patrice Roussel, Bret Singhal, Ronak; Toll, and K.S. Venkatraman. "The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology." *Intel Technology Journal, 8(1).*

[BYF09] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. "Analyzing CUDA workloads using a detailed GPU simulator." In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, pp. 163–174, 2009.

[CB95] Tien-Fu Chen and Jean-Loup Baer. "Effective hardware-based data prefetching for high-performance processors." *Computers, IEEE Transactions on*, **44**:609–623, (May 1995).

[CBM09a] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, 2009.

[CBM09b] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, (Oct. 2009.

[CLL10] Hsiang-Yun Cheng, Chung-Hsiang Lin, Jian Li, and Chia-Lin Yang. "Memory Latency Reduction via Thread Throttling." In *Proceedings of the 2010 43rd*

Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '43, pp. 53–64, Washington, DC, USA, 2010. IEEE Computer Society.

[CTY13] Jianmin Chen, Xi Tao, Zhen Yang, Jih-Kwon Peir, Xiaoyuan Li, and Shih-Lien Lu. "Guided Region-Based GPU Scheduling: Utilizing Multi-thread Parallelism to Hide Memory Latency." In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 441–451, 2013.

[Dav06] Norman P. Jouppi David Tarjan, Shyamkumar Thoziyoor. "CACTI 4.0 Technical Report." In *HP Laboratories Palo Alto*, 2006.

[DMM10] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. "The Scalable Heterogeneous Computing SHOC Benchmark Suite." In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pp. 63–74, New York, NY, USA, 2010. ACM.

[EML09] E. Ebrahimi, O. Mutlu, Chang Joo Lee, and Y.N. Patt. "Coordinated control of multiple prefetchers in multi-core systems." In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 316–326, (Dec. 2009).

[EMP09] E. Ebrahimi, O. Mutlu, and Y.N. Patt. "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems." In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pp. 7–17, (Feb. 2009).

[FJ94] K. I. Farkas and N. P. Jouppi. "Complexity/Performance Tradeoffs with Non-blocking Loads." In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, pp. 211–222, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[FPJ92] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. "Stride directed prefetching in scalar processors." In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, pp. 102–110, (1992).

[GBK09] 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 Comput. Archit. Lett.*, **8**(1):25–28, January 2009.

[GJT11] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. "Energy-efficient mechanisms for managing thread context in throughput processors." In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pp. 235–246, 2011.

[HFL08] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. "Mars: A MapReduce Framework on Graphics Processors." In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pp. 260–269, 2008.

[HM94] L. Harrison and S. Mehrotra. "A Data Prefetch Mechanism for Accelerating General Computation." Technical report, University of Illinois at Urbana-Cahmpaing, (May 1994).

[ISK04] Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G. Abraham. "Effective stream-based and execution-based data prefetching." In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pp. 1–11, New York, NY, USA, (2004).

[JG97] Doug Joseph and Dirk Grunwald. "Prefetching using Markov predictors." In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pp. 252–263, New York, NY, USA, 1997.

[JKC13] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance." In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pp. 395–406, 2013.

[JKM13] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. "Orchestrated Scheduling and Prefetching for GPGPUs." In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pp. 332–343, 2013.

[Jou90] N.P. Jouppi. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers." In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pp. 364–373, (May 1990).

[KDK11] 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, 2011.

[KJK12] Onur Kayiran, Adwait Jog, Mahmut T. Kandermir, and Chita R. Das. "Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs." Technical report, 2012.

[Kro98] David Kroft. "Lockup-free Instruction Fetch/Prefetch Cache Organization." In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, pp. 195–201, 1998.

[KW12] D. Kirk and Wen mei. W. Hwu. "Programming Massively Parallel Processors, 2nd edition.", 2012.

[LK10] Nagesh N. Lakshminarayana and Hyesoon Kim. "Effect of Instruction Fetch and Memory Scheduling on GPU Performance." In *Workshop on Language, Compiler, and Architecture Support for GPGPU*, pp. 128–138, New York, NY, USA, 2010.

[LLK10] Jaekyu Lee, N.B. Lakshminarayana, Hyesoon Kim, and R. Vuduc. "Many-Thread Aware Prefetching Mechanisms for GPGPU Applications." In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pp. 213–224, (December 2010).

[LNO08] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. "NVIDIA Tesla: A Unified Graphics and Computing Architecture." *Micro, IEEE*, **28**(2):39–55, 2008.

[LSM14] Minseok Lee, Seokwoo Song, Joosik Moon, J. Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. "Improving GPGPU resource utilization through alternative thread block scheduling." In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 260–271, Feb 2014.

[Mun13] Aaftab Munshi. "The OpenCL Specification.", 2013.

[NBC14] C. Nugteren, G. J. van den Braak, H. Corporaal, and H. Bal. "A detailed GPU cache model based on reuse distance theory." In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 37–48, Feb 2014.

[NDS04] Kyle J. Nesbit, Ashutosh S. Dhodapkar, and James E. Smith. "AC/DC: An Adaptive Data Cache Prefetcher." In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pp. 135–145, Washington, DC, USA, (2004).

[NS04] K.J. Nesbit and J.E. Smith. "Data Cache Prefetching Using a Global History Buffer." In *Software, IEE Proceedings-*, (February 2004).

[NSL11] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. "Improving GPU Performance via Large Warps and Two-level Warp Scheduling." In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 308–317, 2011.

[NVIa] NVIDIA. "CUDA SDK V2.3.".

[NVIb] NVIDIA. "Geforce 8800 graphics processors." http://www.nvidia.com/page/geforce 8800.html.

[NVIc] NVIDIA. "Nvidia GeForce GTX 480 specifications." http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications.

[NVI09] NVIDIA. "Fermi: Nvidia's next generation cuda compute architecture." http://www.nvidia.com/fermi, 2009.

[NVI11a] NVIDIA. "CUDA C/C++ SDK Code Samples v4.0.", 2011.

[NVI11b] NVIDIA. "NVIDIA CUDA Programming Guide 4.0.", 2011.

[PK94] S. Palacharla and R. E. Kessler. "Evaluating stream buffers as a secondary cache replacement." In *Proceedings of the 21st annual international symposium on Computer architecture*, ISCA '94, pp. 24–33, Los Alamitos, CA, USA, 1994.

[RDK00] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. "Memory Access Scheduling." In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pp. 128–138, 2000.

[ROA12] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. "Cache-Conscious Wavefront Scheduling." In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pp. 72–83, 2012.

[ROA13] 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, pp. 99–110, 2013.

[RRS08] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. "Program optimization space pruning for a multithreaded gpu." In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pp. 195–204, New York, NY, USA, (2008).

[SD88] C. Scheurich and M. Dubois. "The Design of a Lockup-free Cache for High-performance Multiprocessors." In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, pp. 352–359, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[SF91] Gurindar S. Sohi and Manoj Franklin. "High-bandwidth Data Memory Systems for Superscalar Processors." In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pp. 53–62, New York, NY, USA, 1991. ACM.

[SKT05] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. "POWER5 system microarchitecture." *IBM Journal of Research and Development*, **49**:505–521, (July 2005).

[SMK07] S. Srinath, O. Mutlu, Hyesoon Kim, and Y.N. Patt. "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers." In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 63–74, (Feb. 2007).

[SQP08] 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, pp. 277–286, New York, NY, USA, 2008. ACM.

[SRS12] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing." In *Technical Report IMPACT-12-01, University of Illinois, at Urbana-Champaign*, 2012.

[TCT06] James Tuck, Luis Ceze, and Josep Torrellas. "Scalable Cache Miss Handling for High Memory-Level Parallelism." In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pp. 409–422, Washington, DC, USA, 2006. IEEE Computer Society.

[WPS10] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. "Demystifying GPU microarchitecture through microbenchmarking." In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pp. 235–246, 2010.

[YXK10] 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, pp. 86–97, New York, NY, USA, (2010).