Techniques for Shared Resource Management in Systems with Throughput Processors

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Rachata Ausavarungnirun
M.S., Electrical & Computer Engineering, Carnegie Mellon University
B.S., Electrical & Computer Engineering, Carnegie Mellon University
B.S., Computer Science, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

May, 2017

Acknowledgements

First and foremost, I would like to thank my parents, Khrieng and Ruchanee Ausavarungnirun for their endless encouragement, love, and support. In addition to my family, I would like to thank my advisor, Prof. Onur Mutlu, for providing me with great research environment. He taught me many important aspects of research and shaped me into the researcher I am today.

I would like to thank all my committee members, Prof. James Hoe, Dr. Gabriel Loh, Prof. Chris Rossbach and Prof. Kayvon Fatahalian, who provided me multiple feedback on my research and spent a lot of their time and effort to help me complete this dissertation. Special thank to Professor James Hoe, my first mentor at CMU, who taught me all the basics since my sophomore year. Professor Hoe introduced me to many interesting research projects within CALCM. Thanks to Dr. Gabriel Loh for his guidance, which helped me tremendously during the first four years of my PhD. Thanks to Prof. Chris Rossbach for being a great mentor, providing me with guidance, feedback and support for my research. Both Dr. Loh and Prof. Rossbach provided me with lots of real-world knowledge from the industry, which further enhanced the quality of my research. Lastly, thanks to Prof. Kayvon Fatahalian for his knowledge and valuable comments on my GPU research.

All members of SAFARI have been like a family to me. This dissertation is done thanks to lots of support and feedback from them. Donghyuk Lee has always been a good friend and a great mentor. His work ethic is something I always look up to. Thanks to Kevin Chang for all the valuable feedback throughout my PhD. Thanks to Yoongu Kim and Lavanya Subramanian for teaching me on several DRAM-related topics. Thanks to Samira Khan and Saugata Ghose for their guidance. Thanks to Hongyi Xin and Yixin Luo for their positive attitudes and their friendship. Thanks to Vivek Seshadri and Gennady Pekhimenko for their research insights. Thanks to Chris Fallin and Justin Meza for all the helps, especially during the early years of my PhD. They provided tremendous help when I am preparing for my qualification exam. Thanks to Nandita Vijaykumar for all GPU-related discussions. Thanks to Hanbin Yoon, Jamie Liu, Ben Jaiyen, Chris Craik, Kevin Hsieh, Yang Li, Amirali Bouroumand, Jeremie Kim, Damla Senol and Minesh Patel for all their interesting research discussions.

In additional to people in the SAFARI group, I would like to thank Onur Kayiran and Adwait Jog, who have been great colleagues and have been providing me with valuable discussions on various GPU-related research topics. Thanks to Mohammad Fattah for a great collaboration on Network-on-chip. Thanks to Prof. Reetu Das for her inputs on my Network-on-chip research projects. Thanks to Eriko Nurvitadhi and Peter Milder, both of whom were my mentors during my undergrad years. Thanks to John and Claire Bertucci for their fellowship support. Thanks to Dr. Pattana Wangaryattawanich and Dr. Polakit Teekakirikul for their friendship and mental support. Thanks to several members of the Thai Scholar community as well as several members of the Thai

community in Pittsburgh for their friendship. Thanks to support from AMD, Facebook, Google, IBM, Intel, Microsoft, NVIDIA, Qualcomm, VMware, Samsung, SRC, and support from NSF grants numbers 0953246, 1065112, 1147397, 1205618, 1212962, 1213052, 1302225, 1302557, 1317560, 1320478, 1320531, 1409095, 1409723, 1423172, 1439021 and 1439057.

Lastly, I would like to give a special thank to my wife, Chatchanan Doungkamchan for her endless love, support and encouragement. She understands me and helps me with every hurdle I have been through. Her work ethic and the care she gives to her research motivate me to work harder to become a better researcher. She provides me with the perfect environment that allows me to focus on improving myself and my work while trying to make sure neither of us are burned-out from over working. I could not have completed any of the works done in this dissertation without her support.

Abstract

The continued growth of the computational capability of throughput processors has made throughput processors the platform of choice for a wide variety of high performance computing applications. Graphics Processing Units (GPUs) are a prime example of throughput processors that can deliver high performance for applications ranging from typical graphics applications to general-purpose data parallel (GPGPU) applications. However, this success has been accompanied by new performance bottlenecks throughout the memory hierarchy of GPU-based systems. This dissertation identifies and eliminates performance bottlenecks caused by major sources of interference throughout the memory hierarchy.

Specifically, we provide an in-depth analysis of inter- and intra-application as well as inter-address-space interference that significantly degrade the performance and efficiency of GPU-based systems.

To minimize such interference, we introduce changes to the memory hierarchy for systems with GPUs that allow the memory hierarchy to be aware of both CPU and GPU applications' characteristics. We introduce mechanisms to dynamically analyze different applications' characteristics and propose four major changes throughout the memory hierarchy.

First, we introduce Memory Divergence Correction (MeDiC), a cache management mechanism that mitigates intra-application interference in GPGPU applications by allowing the shared L2 cache and the memory controller to be aware of the GPU's warp-level memory divergence characteristics. MeDiC uses this warp-level memory divergence information to give more cache space and more memory bandwidth to warps that benefit most from utilizing such resources. Our evaluations show that MeDiC significantly outperforms multiple state-of-the-art caching policies proposed for GPUs.

Second, we introduce the Staged Memory Scheduler (SMS), an application-aware CPU-GPU memory request scheduler that mitigates inter-application interference in heterogeneous CPU-GPU systems. SMS creates a fundamentally new approach to memory controller design that decouples the memory controller into three significantly simpler structures, each of which has a separate task, These structures operate together to greatly improve both system performance and fairness. Our three-stage memory controller first groups requests based on row-buffer locality. This grouping allows the second stage to focus on inter-application scheduling decisions. These two stages enforce high-level policies regarding performance and fairness. As a result, the last stage is simple logic that deals only with the low-level DRAM commands and timing. SMS is also configurable: it allows the system software to trade off between the quality of service provided to the CPU versus GPU applications. Our evaluations show that SMS not only reduces inter-application interference caused by the GPU, thereby improving heterogeneous system performance, but also provides better scalability and power efficiency compared to multiple state-of-the-art memory schedulers.

Third, we redesign the GPU memory management unit to efficiently handle new problems caused by the massive address translation parallelism present in GPU computation units in multi-GPU-application environments. Running multiple GPGPU applications concurrently induces significant inter-core thrashing on the shared address translation/protection units; e.g., the shared Translation Lookaside Buffer (TLB), a new phenomenon that we call *inter-address-space interference*. To reduce this interference, we introduce *Multi Address Space Concurrent Kernels* (MASK). MASK introduces TLB-awareness throughout the GPU memory hierarchy and introduces TLB- and cache-bypassing techniques to increase the effectiveness of a shared TLB.

Finally, we introduce Mosaic, a hardware-software cooperative technique that further increases the effectiveness of TLB by modifying the memory allocation policy in the system software. Mosaic introduces a high-throughput method to support large pages in multi-GPU-application environments. The key idea is to ensure memory allocation preserve address space contiguity to allow pages to be coalesced without any data movements. Our evaluations show that the MASK-Mosaic combination provides a simple mechanism that eliminates the performance overhead of address translation in GPUs without significant changes to GPU hardware, thereby greatly improving GPU system performance.

The key conclusion of this dissertation is that a combination of GPU-aware cache and memory management techniques can effectively mitigate the memory interference on current and future GPU-based systems as well as other types of throughput processors.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Throughput processor is a type of processors that consists of numerous simple processing cores. Throughput processor allows applications to achieve very high throughput by executing a massive number of compute operations on these processing cores in parallel within a single cycle [5, 7, 8, 14, 27, 43, 54, 71, 74, 77, 136, 146, 164, 166, 249, 268, 269, 271, 272, 273, 276, 304, 323, 326, 327, 343, 344, 365, 366, 379, 384]. These throughput processors incorporate a variety of processing paradigms, such as vector processors, which utilize a specific execution model called Single Instruction Multiple Data (SIMD) model that allows one instruction to be operated on multiple data [43, 74, 77, 146, 323, 326, 327, 343], processors that utilize a technique called fine-grained multithreading, which allows the processor to issue instructions from different threads after every cycle [14, 27, 136, 344, 365, 366], or processors that utilize both techniques [5, 7, 8, 54, 71, 164, 166, 249, 268, 269, 271, 272, 273, 276, 304, 379, 384]. One of the most prominent throughput processors available in modern day computing systems that utilize both SIMD execution model and fine-grained multithreading is the Graphics Processing Units (GPUs). This dissertation uses GPUs as an example class of throughput processors.

GPUs have enormous parallel processing power due to the large number of computational units they provide. Modern GPU programming models exploit this processing power using a large amount of thread-level parallelism. GPU applications can be broken down into thousands

of threads, allowing GPUs to use an execution model called SIMT (Single Instruction Multiple Thread), which enables the GPU cores to tolerate dependencies and long memory latencies. The thousands of threads within a GPU application are clustered into *work groups* (or *thread blocks*), where each thread block consists of a collection of threads that are run concurrently. Within a thread block, threads are further grouped into smaller units, called *warps* [226] or *wavefronts* [11]. Every cycle, each GPU core executes a single warp. Each thread in a warp executes the same instruction (i.e., is at the same program counter) in lockstep, which is an example of the *SIMD* (Single Instruction, Multiple Data) [106] execution model. This highly-parallel SIMT/SIMD execution model allows the GPU to complete several hundreds to thousands of operations every cycle.

GPUs are present in many modern systems. These GPU-based systems range from traditional discrete GPUs [7, 8, 249, 271, 272, 273, 276, 304, 379] to heterogeneous CPU-GPU architectures [5, 54, 71, 164, 166, 249, 268, 269, 304, 384]. In all of these systems with GPUs, resources throughout the memory hierarchy, e.g., core-private and shared caches, main memory, the interconnects, and the memory management units are shared across multiple threads and processes that execute concurrently in both the CPUs and the GPUs.

1.1. Resource Contention and Memory Interference Problem in Systems with GPUs

Due to the limited shared resources in these systems, applications oftentimes are not able to achieve their ideal throughput (as measured by, e.g., computed instructions per cycle). Shared resources become the bottleneck and create inefficiency because accesses from one thread or application can interfere with accesses from other threads or applications in any shared resources, leading to both bandwidth and space contention, resulting in lower performance. *The main goal of this dissertation is to analyze and mitigate the major memory interference problems throughout shared resources in the memory hierarchy of current and future systems with GPUs.*

We focus on three major types of memory interference that occur in systems with GPUs: 1) intra-application interference among different GPU threads, 2) inter-application interference that is

caused by both CPU and GPU applications, and 3) inter-address-space interference during address translation when multiple GPGPU applications concurrently share the GPUs.

Intra-application interference is a type of interference that originates from GPU threads within the *same* GPU application. When a GPU executes a GPGPU application, the threads that are scheduled to run on the GPU cores execute concurrently. Even though these threads belong to the same kernel, they contend for shared resources, causing interference to each other [36, 69, 70, 222]. This intra-application interference leads to the significant slowdown of threads running on GPU cores and lowers the performance of the GPU.

Inter-application interference is a type of interference that is caused by concurrently-executing CPU and GPU applications. It occurs in systems where a CPU and a GPU share the main memory system. This type of interference is especially observed in recent heterogeneous CPU-GPU systems [33, 54, 55, 71, 161, 163, 164, 166, 171, 189, 191, 249, 268, 304, 384], which introduce an integrated Graphics Processing Unit (GPU) on the same die with the CPU cores. Due to the GPU's ability to execute a very large number of parallel threads, GPU applications typically demand significantly more memory bandwidth than typical CPU applications. Unlike GPU applications that are designed to tolerate the long memory latency by employing massive amounts of multithreading [7, 8, 9, 33, 54, 71, 164, 166, 249, 268, 269, 271, 272, 273, 276, 304, 379, 384], CPU applications typically have much lower tolerance to latency [33, 93, 197, 198, 209, 260, 261, 353, 354, 355, 357]. The high bandwidth consumption of the GPU applications heavily interferes with the progress of other CPU applications that share the same hardware resources.

Inter-address-space interference arises due to the address translation process in an environment where multiple GPU applications share the same GPU, e.g., a shared GPU in a cloud infrastructure. We discover that when multiple GPGPU applications concurrently use the same GPU, the address translation process creates additional contention at the shared memory hierarchy, including the Translation Lookaside Buffers (TLBs), caches, and main memory. This particular type of interference can cause a severe slowdown to all applications and the system when multiple GPGPU applications are concurrently executed on a system with GPUs.

While previous works propose mechanisms to reduce interference and improve the performance of GPUs (See Chapter 3 for a detailed analyses of these previous works), these approaches 1) focus only on a subset of the shared resources, such as the shared cache or the memory controller and 2) generally do not take into account the characteristics of the applications executing on the GPUs.

1.2. Thesis Statement and Our Overarching Approach: Application Awareness

With the understanding of the causes of memory interference, our thesis statement is that **a combination of GPU-aware cache and memory management techniques can mitigate memory interference caused by GPUs on current and future systems with GPUs.** To this end, we propose to mitigate memory interference in current and future GPU-based systems via GPU-aware and GPU-application-aware resource management techniques. We redesign the memory hierarchy such that *each component in the memory hierarchy is aware of the GPU applications' characteristics.* The key idea of our approach is to extract important features of different applications in the system and use them in managing memory hierarchy resources much more intelligently. These key features consist of, but are not limited to, memory access characteristics, utilization of the shared cache, usage of shared main memory and demand for the shared TLB. Exploiting these features, we introduce modifications to the shared cache, the memory request scheduler, the shared TLB and the GPU memory allocator to reduce the amount of inter-application, intra-application and inter-address-space interference based on applications' characteristics. We give a brief overview of our major new mechanisms in the rest of this section.

1.2.1. Minimizing Intra-application Interference

Intra-application interference occurs when multiple threads in the GPU contend for the shared cache and the shared main memory. Memory requests from one thread can interfere with memory requests from other threads, leading to low system performance. As a step to reduce this intra-

application interference, we introduce Memory Divergence Correction (MeDiC) [36], a cache and memory controller management scheme that is designed to be aware of different types of warps that access the shared cache, and selectively prioritize warps that benefit the most from utilizing the cache. This new mechanism first characterizes different types of warps based on how much benefit they receive from the shared cache. To effectively characterize warp-type, MeDiC uses the memory divergence patterns, i.e., the diversity of how long each load and store instructions in the warp takes. We observe that GPGPU applications exhibit different levels of heterogeneity in their memory divergence behavior at the shared L2 cache within the GPU. As a result, (1) some warps benefit significantly from the cache, while others make poor use of it; (2) the divergence behavior of a warp tends to remain stable for long periods of the warp's execution; and (3) the impact of memory divergence can be amplified by the high queuing latencies at the L2 cache.

Based on the heterogeneity in warps' memory divergence behavior, we propose a set of techniques, collectively called *Memory Divergence Correction* (MeDiC), that reduce the negative performance impact of memory divergence and cache queuing. MeDiC uses warp divergence characterization to guide three warp-aware components in the memory hierarchy: (1) a cache bypassing mechanism that exploits the latency tolerance of warps that do not benefit from using the cache, to both alleviate queuing delay and increase the hit rate for warps that benefit from using the cache, (2) a cache insertion policy that prevents data from warps that benefit from using the cache from being prematurely evicted, and (3) a memory controller that prioritizes the few requests received from warps that benefit from using the cache, to minimize stall time. Our evaluation shows that MeDiC is effective at exploiting inter-warp heterogeneity and delivers significant performance and energy improvements over the state-of-the-art GPU cache management technique [222].

1.2.2. Minimizing Inter-application Interference

Inter-application interference occurs when multiple processor cores (CPUs) and a GPU integrated together on the same chip share the off-chip DRAM (and perhaps some caches). In such as system, requests from the GPU can heavily interfere with requests from the CPUs, leading to low

system performance and starvation of cores. Even though previously-proposed application-aware memory scheduling policies designed for CPU-only scenarios (e.g., [93, 197, 198, 209, 260, 261, 317, 353, 354, 355, 357]) can be applied on a CPU-GPU heterogeneous system, we observe that the GPU requests occupy a significant portion of request buffer space and thus reduce the visibility of CPU cores' requests to the memory controller, leading to lower system performance. Increasing the request buffer space requires complex logic to analyze applications' characteristics, assign priorities for each memory request and enforce these priorities when the GPU is present. As a result, these past proposals for application-aware memory scheduling in CPU-only systems can perform poorly on a CPU-GPU heterogeneous system at low complexity (as we show in this dissertation).

To minimize the inter-application interference in CPU-GPU heterogeneous systems, we introduce a new memory controller called the Staged Memory Scheduler (SMS) [33], which is both application-aware and GPU-aware. Specifically, SMS is designed to facilitate GPU applications' high bandwidth demand, improving performance and fairness significantly. SMS introduces a fundamentally new approach that decouples the three primary tasks of the memory controller into three significantly simpler structures that together improve system performance and fairness. The three-stage memory controller first groups requests based on row-buffer locality in its first stage, called the *Batch Formation stage*. This grouping allows the second stage, called the *Batch Scheduler* stage, to focus mainly on inter-application scheduling decisions. These two stages collectively enforce high-level policies regarding performance and fairness, and therefore the last stage can get away with using simple per-bank FIFO queues (no further command reordering within each bank) and straight forward logic that deals only with the low-level DRAM commands and timing. This last stage is called the DRAM Command Scheduler stage.

Our evaluation shows that SMS is effective at reducing inter-application interference. SMS delivers superior performance and fairness compared to state-of-the-art memory schedulers [197, 198, 261, 317], while providing a design that is significantly simpler to implement and that has significantly lower power consumption.

1.2.3. Minimizing Inter-address-space Interference

Inter-address-space interference occurs when the GPU is *shared* among multiple GPGPU applications in large-scale computing environments [9, 32, 174, 271, 272, 273, 276, 376]. Much of the inter-address-space interference problem in a contemporary GPU lies within the memory system, where multi-application execution requires virtual memory support to manage the address spaces of each application and to provide memory protection. We observe that when multiple GPGPU applications spatially share the GPU, a significant amount of inter-core thrashing occurs on the shared TLB within the GPU. We observe that this contention at the shared TLB is high enough to prevent the GPU from successfully hiding memory latencies, which causes TLB contention to become a first-order performance concern.

Based on our analysis of the TLB contention in a modern GPU system executing multiple applications, we introduce two mechanisms. First, we design Multi Address Space Concurrent Kernels (MASK). The key idea of MASK is to 1) extend the GPU memory hierarchy to efficiently support address translation via the use of multi-level TLBs, and 2) use translation-aware memory and cache management techniques to maximize throughput in the presence of inter-address-space contention. MASK uses a novel token-based approach to reduce TLB miss overheads by limiting the number of thread that can use the shared TLB, and its L2 cache bypassing mechanisms and address-space-aware memory scheduling reduce the inter-address-space interference. We show that MASK restores much of the thread-level parallelism that was previously lost due to address translation.

Second, to further minimize the inter-address-space interference, we introduce Mosaic. Mosaic significantly decreases inter-address-space interference at the shared TLB by increasing TLB reach via support for multiple page sizes, including very large pages. To enable multi-page-size support, we provide two *key observations*. First, we observe that the vast majority of memory allocations and memory deallocations are performed *en masse* by GPGPU applications in phases, typically soon after kernel launch or before kernel exit. Second, long-lived memory objects that usually increase fragmentation and induce complexity in CPU memory management are largely absent in

the GPU setting. These two observations make it relatively easy to predict memory access patterns of GPGPU applications and simplify the task of detecting when a memory region can benefit from using large pages.

Based on the prediction of the memory access patterns, Mosaic 1) modifies GPGPU applications' memory layout in system software to preserve address space contiguity, which allows the GPU to splinter and coalesce pages very fast without moving data and 2) periodically performs memory compaction while still preserving address space contiguity to avoid memory bloat and data fragmentation. Our prototype shows that Mosaic is very effective at reducing inter-address-space interference at the shared TLB and limits the number of shared TLB miss rate to less than 1% on average (down from 25.4% in the baseline shared TLB).

In summary, MASK incorporates TLB-awareness throughout the memory hierarchy and introduces TLB- and cache-bypassing techniques to increase the effectiveness of a shared TLB. Mosaic provides a hardware-software cooperative technique that modifies the memory allocation policy in the system software and introduces a high-throughput method to support large pages in multi-GPU-application environments. The MASK-Mosaic combination provides a simple mechanism to eliminate the performance overhead of address translation in GPUs without requiring significant changes in GPU hardware. These techniques work together to significantly improve system performance, IPC throughput, and fairness over the state-of-the-art memory management technique [303].

1.3. Contributions

We make the following major contributions:

- We provide an in-depth analyses of three different types of memory interference in systems with GPUs. Each of these three types of interference significantly degrades the performance and efficiency of the GPU-based systems. To minimize memory interference, we introduce mechanisms to dynamically analyze different applications' characteristics and propose four major changes throughout the memory hierarchy of GPU-based systems.

- We introduce Memory Divergence Correction (MeDiC). MeDiC is a mechanism that minimizes intra-application interference in systems with GPUs. MeDiC is the first work that observes that the different warps within a GPGPU application exhibit heterogeneity in their memory divergence behavior at the shared L2 cache, and that some warps do not benefit from the few cache hits that they have. We show that this memory divergence behavior tends to remain consistent throughout long periods of execution for a warp, allowing for fast, online warp divergence characterization and prediction. MeDiC takes advantage of this warp characterization via a combination of *warp-aware* cache bypassing, cache insertion and memory scheduling techniques. Chapter 4 provides the detailed design and evaluation of MeDiC.

- We demonstrate how the GPU memory traffic in heterogeneous CPU-GPU systems can cause severe inter-application interference, leading to poor performance and fairness. We propose a new memory controller design, the Staged Memory Scheduler (SMS), which delivers superior performance and fairness compared to three state-of-the-art memory schedulers [197,198,317], while providing a design that is significantly simpler to implement. The key insight behind SMS's scalability is that the primary functions of sophisticated memory controller algorithms can be decoupled into different stages in a multi-level scheduler. Chapter 5 provides the design and the evaluation of SMS in detail.

- We perform a detailed analysis of the major problems in state-of-the-art GPU virtual memory management that hinders high-performance multi-application execution. We discover a new type of memory interference, which we call *inter-address-space interference*, that arises from a significant amount of inter-core thrashing on the shared TLB within the GPU. We also discover that the TLB contention is high enough to prevent the GPU from successfully hiding memory latencies, which causes TLB contention to become a first-order performance concern in GPU-based systems. Based on our analysis, we introduce Multi Address Space Concurrent Kernels (MASK). MASK extends the GPU memory hierarchy to efficiently support address translation through the use of multi-level TLBs, and uses translation-aware

memory and cache management to maximize IPC (instruction per cycle) throughput in the presence of inter-application contention. MASK restores much of the thread-level parallelism that was previously lost due to address translation. Chapter 6 analyzes the effect of inter-address-space interference and provides the detailed design and evaluation of MASK.

- To further minimize the inter-address-space interference, we introduce Mosaic. Mosaic further increases the effectiveness of TLB by providing a hardware-software cooperative technique that modifies the memory allocation policy in the system software. Mosaic introduces a low overhead method to support large pages in multi-GPU-application environments. The key idea of Mosaic is to ensure memory allocation preserve address space contiguity to allow pages to be coalesced without any data movements. Our prototype shows that Mosaic significantly increases the effectiveness of the shared TLB in a GPU and further reduces inter-address-space interference. Chapter 7 provides the detailed design and evaluation of Mosaic.

1.4. Dissertation Outline

This dissertation is organized into eight Chapters. Chapter 2 presents background on modern GPU-based systems. Chapter 3 discusses related prior works on resource management, where techniques can potentially be applied to reduce interference in GPU-based systems. Chapter 4 presents the design and evaluation of MeDiC. MeDiC is a mechanism that minimizes *intra-application interference* by redesigning the shared cache and the memory controller to be aware of different types of warps. Chapter 5 presents the design and evaluation of SMS. SMS is a GPU-aware and application-aware memory controller design that minimizes the *inter-application interference*. Chapter 6 presents a detailed analysis of the performance impact of *inter-address-space interference*. It then proposes MASK, a mechanism that minimizes *inter-address-space interference* by introducing TLB-awareness throughout the memory hierarchy. Chapter 7 presents the design for Mosaic. Mosaic provides a hardware-software cooperative technique that reduces *inter-address-space interference* by lowering contention at the shared TLB. Chapter 8 provides the summary of

common principles and lessons learned. Chapter 9 provides the summary of this dissertation as well as future research directions that are enabled by this dissertation.

Chapter 2

The Memory Interference Problem in Systems with GPUs

We first provide background on the architecture of a modern GPU, and then we discuss the bottlenecks that highly-multithreaded applications can face either when executed alone on a GPU or when executing with other CPU or GPU applications.

2.1. Modern Systems with GPUs

In this section, we provide a detailed explanation of the GPU architecture that is available on modern systems. Section 2.1 discusses a typical modern GPU architecture [5, 7, 8, 54, 71, 164, 166, 249, 268, 269, 271, 272, 273, 276, 304, 379, 384] as well as its memory hierarchy. Section 2.2 discusses the design of a modern CPU-GPU heterogeneous architecture [54, 164, 166, 384] and its memory hierarchy. Section 2.3 discusses the memory management unit and support for address translation.

2.1.1. GPU Core Organization

A typical GPU consists of several GPU cores called *shader cores* (sometimes called *streaming multiprocessors*, or SMs). As shown in Figure 2.1, a GPU core executes SIMD-like instruc-

tions [106]. Each SIMD instruction can potentially operate on multiple pieces of data in parallel. Each data piece operated on by a different thread of control. Hence, the name SIMT (Single Instruction Multiple Thread). Multiple threads that are the same are grouped into a warp. A warp is a collection of threads that are executing the same instruction (i.e., are at the same Program Counter). Multiple warps are grouped into a thread block. Every cycle, a GPU core fetches an available warp (a warp is available if none of its threads are stalled), and issues an instruction associated with those threads (in the example from Figure 2.1, this instruction is from *Warp D* and the address of this instruction is $0x12F2$). In this way, a GPU can potentially retire as many instructions as the number of cores multiplied by the number of threads per warp, enabling high instruction-per-cycle (IPC) throughput. More detail on GPU core organization can be found in [41, 110, 111, 118, 138, 243, 340, 388].

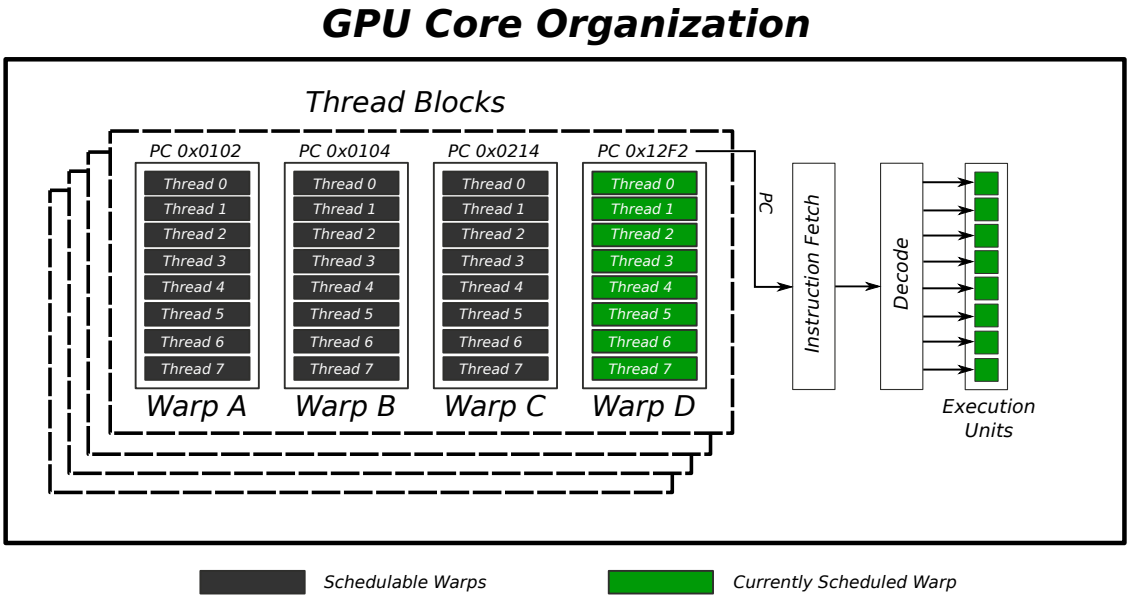


Figure 2.1. Organization of threads, warps, and thread blocks.

2.1.2. GPU Memory Hierarchy

When there is a load or store instruction that needs to access data from the main memory, the GPU core sends a memory request to the memory hierarchy, which is shown in Figure 2.2. This hierarchy typically contains a private data cache, and an interconnect (typically a crossbar) that

connects all the cores with the shared data cache. If the target data is present neither in the private nor the shared data cache, a memory request is sent to the main memory in order to retrieve the data.

GPU Cache Organization and Key Assumptions. Each core has its own private L1 data, texture, and constant caches, as well as a software-managed scratchpad memory [8, 11, 226, 271, 272, 273, 276, 377]. In addition, the GPU also has several shared L2 cache slices and memory controllers. Because there are several methods to design the GPU memory hierarchy, we assume the baseline that decouples the memory channels into multiple memory partitions. A *memory partition unit* combines a single L2 cache slice (which is banked) with a designated memory controller that connects the GPU to off-chip main memory. Figure 2.2 shows a simplified view of how the cores (or SMs), caches, and memory partitions are organized in our baseline GPU.

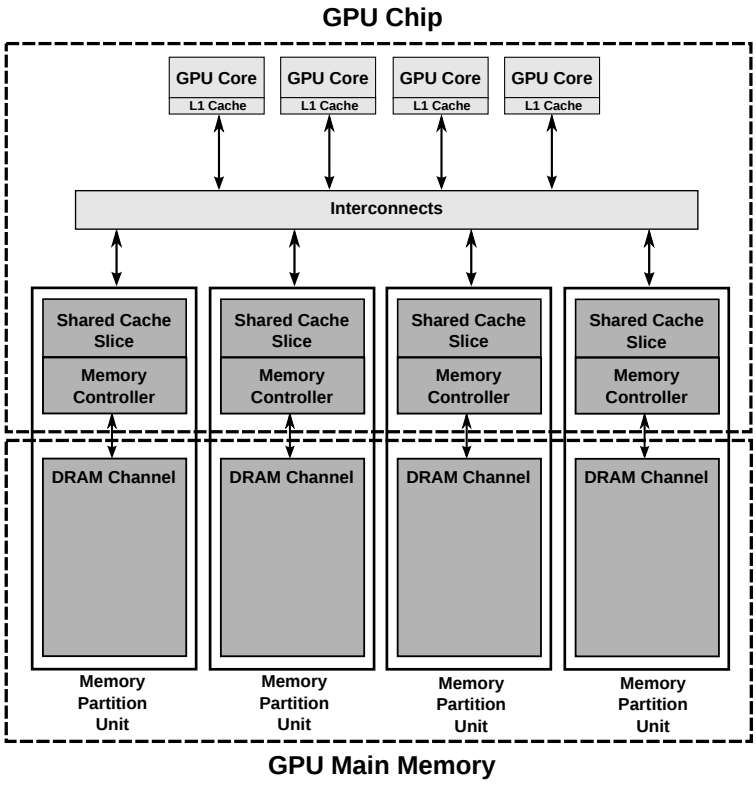


Figure 2.2. Overview of a modern GPU architecture.

GPU Main Memory Organization. Similar to systems with CPUs, a GPU uses DRAM (organized as hierarchical two-dimensional arrays of bitcells) as main memory. Reading or writing

data to DRAM requires that a row of bitcells from the array first be read into a row buffer. This is required because the act of reading the row destroys the row's contents, and so a copy of the bit values must be kept (in the row buffer). Reads and writes operate directly on the row buffer. Eventually, the row is "closed" whereby the data in the row buffer is written back into the DRAM array. Accessing data already loaded in the row buffer, also called a row buffer hit, incurs a shorter latency than when the corresponding row must first be "opened" from the DRAM array. A modern memory controller (memory controller) must orchestrate the sequence of commands to open, read, write and close rows. Servicing requests in an order that increases row-buffer hit rate tends to improve overall throughput by reducing the average latency to service requests. The memory controller is also responsible for enforcing a wide variety of timing constraints imposed by modern DRAM standards (e.g., DDR3) such as limiting the rate of page-open operations (t_{FAW}) and ensuring a minimum amount of time between writes and reads (t_{WTR}). More detail on timing constraints and DRAM operation can be found in [63,64,142,143,199,213,214,215,216,228,331].

Each two-dimensional array of DRAM cells constitutes a bank, and a group of banks forms a rank. All banks within a rank share a common set of command and data buses, and the memory controller is responsible for scheduling commands such that each bus is used by only one bank at a time. Operations on multiple banks may occur in parallel (e.g., opening a row in one bank while reading data from another bank's row buffer) so long as the buses are properly scheduled and any other DRAM timing constraints are honored. A memory controller can improve memory system throughput by scheduling requests such that bank-level parallelism or BLP (i.e., the number of banks simultaneously busy responding to commands) is higher [212, 261]. A memory system implementation may support multiple independent memory channels (each with its own ranks and banks) [37,257] to further increase the number of memory requests that can be serviced at the same time. A key challenge in the implementation of modern, high-performance memory controllers is to effectively improve system performance by maximizing both row-buffer hits and BLP while simultaneously providing fairness among multiple CPUs and the GPU [33].

Key Assumptions. We assume the memory controller consists of a centralized memory request

buffer. Additional details of the memory controller design can be found in Sections 4.4, 6.5 and 7.4.

2.1.3. Intra-application Interference within GPU Applications

While many GPGPU applications can tolerate a significant amount of memory latency due to their parallelism through the SIMT execution model, many previous works (e.g., [41, 65, 110, 111, 138, 175, 176, 243, 264, 318, 319, 378, 388]) observe that GPU cores often stall for a significant fraction of time. One significant source of these stalls is the contention at the shared GPU memory hierarchy [36, 65, 175, 176, 189, 243, 264, 318, 378]. The large amount of parallelism in GPU-based systems creates a significant amount of contention on the GPU's memory hierarchy. Even through all threads in the GPU execute the codes from the same application, data accesses from one warp can interfere with data accesses from other warps. This interference comes in several forms such as additional cache thrashing and queuing delays at both the shared cache and shared main memory. These combine to lower the performance of GPU-based systems. We call this interference the *intra-application interference*.

Memory divergence, where the threads of a warp reach a memory instruction, and some of the threads' memory requests take longer to service than the requests from other threads [36, 65, 243, 264], further exacerbates the effect of *intra-application interference*. Since all threads within a warp operate in lockstep due to the SIMD execution model, the warp cannot proceed to the next instruction until the *slowest* request within the warp completes, and *all* threads are ready to continue execution.

Chapter 4 provides detailed analyses on how to reduce *intra-application interference* at the shared cache and the shared main memory.

2.2. GPUs in CPU-GPU Heterogeneous Architectures

Aside from using off-chip discrete GPUs, modern architectures integrate Graphics Processors integrate a GPU on the same chip as the CPU cores [33, 54, 55, 71, 161, 163, 164, 166, 171, 191, 249, 268, 304, 384]. Figure 2.3 shows the design of these recent heterogeneous CPU-GPU architecture.

As shown in Figure 2.3, parts of the memory hierarchy are being shared across both CPU and GPU applications.

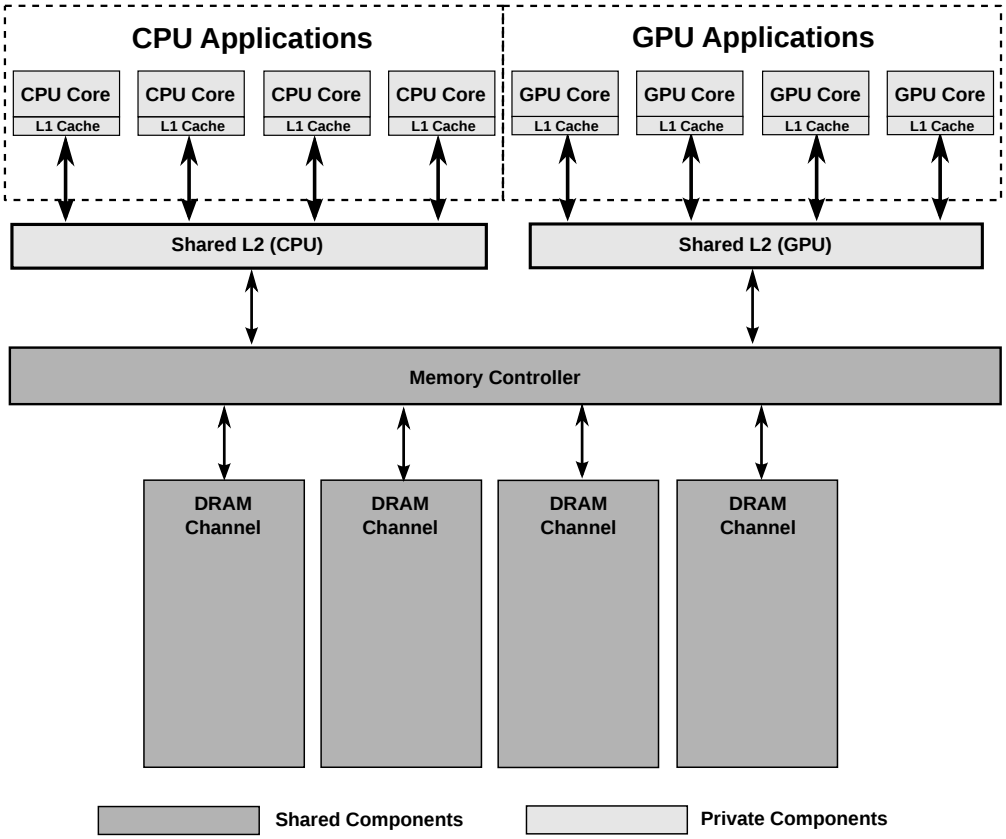


Figure 2.3. The memory hierarchy of a heterogeneous CPU-GPU architecture.

Key Assumptions. We make two key assumptions for the design of heterogeneous CPU-GPU systems. First, we assume that the GPUs and the CPUs do not share the last level caches. Second, we assume that the memory controller is the first point in the memory hierarchy that CPU applications and GPU applications share resources. We applied multiple memory scheduler designs as the baseline for our evaluations. Additional details of these baseline design can be found in Sections 5.3.5 and 5.4.

2.2.1. Inter-application Interference across CPU and GPU Applications

As illustrated in Figure 2.3, the main memory is a major shared resource among cores in modern chip multiprocessor (CMP) systems. Memory requests from multiple cores interfere with each

other at the main memory and create *inter-application interference*, which is a significant impediment to individual application and system performance. Previous works on CPU-only application-aware memory scheduling [93, 197, 198, 209, 260, 261, 353] have addressed the problem by being aware of application characteristics at the memory controller and prioritizing memory requests to improve system performance and fairness. This approach of application-aware memory request scheduling has provided good system performance and fairness in multicore systems.

As opposed to CPU applications, GPU applications are not very latency sensitive as there are a large number of independent threads to cover long memory latencies. However, the GPU requires a significant amount of bandwidth far exceeding even the most memory-intensive CPU applications. As a result, a GPU memory scheduler [226, 272, 276] typically needs a large request buffer that is capable of request coalescing (i.e., combining multiple requests for the same block of memory into a single combined request [226]). Furthermore, since GPU applications are bandwidth intensive, often with streaming access patterns, a policy that maximizes the number of row-buffer hits is effective for GPUs to maximize overall throughput. Hence, a memory scheduler that can improve the effective DRAM bandwidth such as the FR-FCFS scheduler with a large request buffer [41, 317, 397, 406] tends to perform well for GPUs.

This conflicting preference between CPU applications and GPU applications (CPU applications benefit from lower memory request latency while GPU applications benefit from higher DRAM bandwidth) further complicates the design of memory request scheduler for CPU-GPU heterogeneous systems. A design that favors lowering the latency of CPU requests is undesirable for GPU applications while a design that favors providing high bandwidth is undesirable for CPU applications.

In this dissertation, Chapter 5 provides an in-depth analysis of this *inter-application interference* and provides a method to mitigate the interference in CPU-GPU heterogeneous architecture.

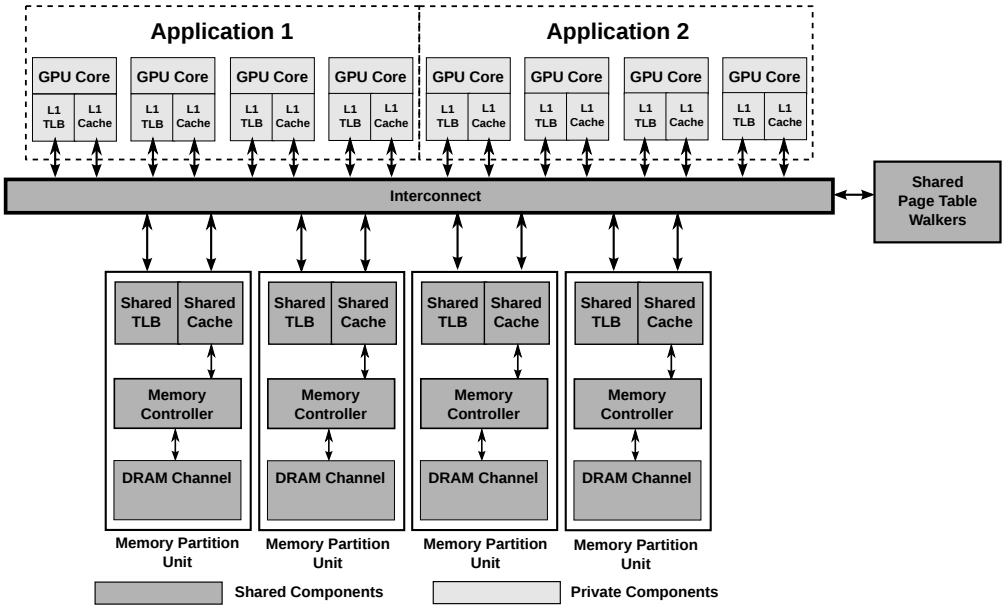


Figure 2.4. A GPU design showing two concurrent GPGPU applications concurrently sharing the GPUs.

2.3. GPUs in Multi-GPU-application Environments

Recently, a newer set of analytic GPGPU applications, such as the Netflix movie recommendation systems [26], or a stock market analyzer [305], require a closely connected, highly virtualized, shared environment. These applications, which benefit from the amount of parallelism GPU provides, do not need to use all resources in the GPU to maximize their performance. Instead, these emerging applications benefit from concurrency - by running a few of these applications together, each sharing some resources on the GPU. NVIDIA GRID [272,276] and AMD FirePro [9] are two examples of spatially share GPU resources across multiple applications.

Figure 2.4 shows the high-level design of how a GPU can be spatially shared across two GPGPU applications. In this example, the GPUs contain multiple shared page table walkers, which are responsible for translating a virtual address into a physical address. This design also contains two level of translation lookaside buffers (TLBs), which cache the virtual-to-physical translation. This design allows the GPU to co-schedule kernels, even applications, concurrently because *address translation enables memory protection across multiple GPGPU applications.*

Key Assumptions. The page table walker can be placed at different locations in the GPU

memory hierarchy. The GPU MMU design proposed by Power et al. places a parallel page table walkers between the private L1 and the shared L2 caches [303]. Other alternative designs place the page table walker at the Input-Output Memory Management Unit (IOMMU), which directly connects to the main memory [5,7,8,54,71,164,166,249,268,269,271,272,273,276,304,304,379, 384], and another GPU MMU design proposed by Cong et al. uses the CPU's page table walker to perform GPU page walks [73]. We found that placing a parallel page table walkers at the shared L2 cache provides the best performance. Hence, we assume the baseline proposed by Power et al. that utilized the per-core private TLB and place the page table walker at the shared L2 cache [303].

2.3.1. Inter-address-space Interference on Multiple GPU Applications

While concurrently executing multiple GPGPU applications that have complementary resource demand can improve GPU utilization, these applications also share two critical resources: the shared address translation unit and the shared TLB. We find that when multiple applications spatially share the GPU, there is a significant amount of thrashing on the shared TLB within the GPU because multiple applications from different address spaces are contending at the shared TLB, the page table walker as well as the shared L2 data cache. We define this phenomenon as the *inter-address-space interference*.

The amount of parallelism on GPUs further exacerbate the performance impact of *inter-address-space interference*. We found that an address translation in response to a single TLB miss typically stalls tens of warps. As a result, a small number of outstanding TLB misses can result in a significant number of warps to become unschedulable, which in turn limits the GPU's most essential latency-hiding capability. We observe that providing address translation in GPUs reduce the GPU performance to 47.3% of the ideal GPU with no address translation, which is a significant performance overhead. As a result, it is even more crucial to mitigate this *inter-address-space interference* throughout the GPU memory hierarchy in multi-GPU-application environments. Chapters 6 and 7 provide detailed design descriptions of the two mechanisms we propose that can be used to reduce this *inter-address-space interference*.

Chapter 3

Related Works on Resource Management in Systems with GPUs

Several previous works have been proposed to address the memory interference problem in systems with GPUs. These previous proposals address certain parts of the main memory hierarchy. In this chapter, we first provide the background on the GPU's execution model. Then, we provide breakdowns of previous works on GPU resource management throughout the memory hierarchy as well as differences between these previous works and techniques presented in this dissertation.

3.1. Background on the Execution Model of GPUs

Modern day GPUs employ two main techniques to enable their parallel processing power: SIMD, which executes multiple data within a single instruction, and fine-grain multithreading, which prevents the GPU cores from stalling by issuing instructions from different threads every cycle. This section provides the background on previous machines and processors that apply similar techniques.

3.1.1. SIMD and Vector Processing

The SIMD execution model, which includes vector processing, been used by several machines in the past. Slotnik et al. in the Solomon Computer [343], Senzig and Smith [327], Crane and Guthens [77], Hellerman [146], CDC 7600 [74], CDC STAR-100 [326], Illiac IV [43] and Cray I [323] are examples of machines that employ a vector processor. In modern systems, Intel MMX [162, 296] and Intel SSE [164] also apply SIMD in order to improve performance. As an alternative of using one instruction to execute multiple data, VLIW [105] generate codes for a parallel machine that allows multiple instructions to operate on multiple data concurrently in a single cycle. Intel i860 [125] and Intel Itanium [240] are examples of processors with the VLIW technology.

3.1.2. Fine-grained Multithreading

Fine-grain multithreading, which is a technique that allows the processor to issue instructions from different threads every cycle, is the key component that enables latency hiding capability in modern day GPUs. CDC 6600 [365, 366], Denelcor HEP [344], MASA [136], APRIL [14] and Tera MTA [27] are examples of machines that utilize fine-grain multithreading.

3.2. Background on Techniques to Reduce Interference of Shared Resources

Several techniques to reduce interference at the shared cache, shared off-chip main memory as well as the shared interconnect have been proposed. In this section, we provide a brief discussion of these works.

3.2.1. Cache Bypassing Techniques

Hardware-based Cache Bypassing Techniques. Several hardware-based cache bypassing mechanisms have been proposed in both CPU and GPU setups. Li et al. propose PCAL, a bypassing mechanism that addresses the cache thrashing problem by throttling the number of threads that time-share the cache at any given time [222]. The key idea of PCAL is to limit the number

of threads that get to access the cache. Li et al. [221] propose a cache bypassing mechanism that allows only threads with high reuse to utilize the cache. The key idea is to use locality filtering based on the reuse characteristics of GPGPU applications, with only high reuse threads having access to the cache. Xie et al. [391] propose a bypassing mechanism at the thread block level. In their mechanism, the compiler statically marks whether thread blocks prefer caching or bypassing. At runtime, the mechanism dynamically selects a subset of thread blocks to use the cache, to increase cache utilization. Chen et al. [69, 70] propose a combined warp throttling and bypassing mechanism for the L1 cache based on the cache-conscious warp scheduler [318]. The key idea is to bypass the cache when resource contention is detected. This is done by embedding history information into the L2 tag arrays. The L1 cache uses this information to perform bypassing decisions, and only warps with high reuse are allowed to access the L1 cache. Jia et al. propose an L1 bypassing mechanism [172], whose key idea is to bypass requests when there is an associativity stall. Dai et al. propose a mechanism to bypass cache based on a model of a cache miss rate [79].

There are also several other CPU-based cache bypassing techniques. These techniques include using additional buffers track cache statistics to predict cache blocks that have high utility based on reuse count [67,96,117,178,193,227,387,398], reuse distance [67,89,104,114,134,287,386,396], behavior of the cache block [169] or miss rate [72, 369]

Software-based Cache Bypassing Techniques. Because GPUs allow software to specify whether to utilize the cache or not [277,278]. Software based cache bypassing techniques have also been proposed to improve system performance. Li et al. [220] propose a compiler-based technique that performs cache bypassing using a method similar to PCAL [222]. Xie et al. [390] propose a mechanism that allows the compiler to perform cache bypassing for global load instructions. Both of these mechanisms apply bypassing to *all* loads and stores that utilize the shared cache, without requiring additional characterization at the compiler level. Mekkat et al. [242] propose a bypassing mechanism for when a CPU and a GPU share the last level cache. Their key idea is to bypass GPU cache accesses when CPU applications are cache sensitive, which is not applicable to GPU-only execution.

3.2.2. Cache Insertion and Replacement Policies

Many works have proposed different insertion policies for CPU systems (e.g., [167, 168, 307, 334]). Dynamic Insertion Policy (DIP) [167] and Dynamic Re-Reference Interval Prediction (DRRIP) [168] are insertion policies that account for cache thrashing. The downside of these two policies is that they are unable to distinguish between high-reuse and low-reuse blocks in the same thread [334]. The Bi-modal Insertion Policy [307] dynamically characterizes the cache blocks being inserted. None of these works on cache insertion and replacement policies [167, 168, 307, 334] take warp type characteristics or memory divergence behavior into account.

3.2.3. Cache and Memory Partitioning Techniques

Instead of mitigating the interference problem between applications by scheduling requests at the memory controller, Awasthi et al. propose a mechanism that spreads data in the same working set across memory channels in order to increase memory level parallelism [37]. Muralidhara et al. propose memory channel partitioning (MCP) to map applications to different memory channels based on their memory-intensities and row-buffer locality to reduce inter-application interference [257]. Mao et al. propose to partition GPU channels and only allow a subset of threads to access each memory channel [239]. In addition to channel partitioning, several works also propose to partition DRAM banks [158, 229, 389] and the shared cache [310, 356] to improve performance. These partitioning techniques are orthogonal to our proposals and can be combined to improve the performance of GPU-based systems.

3.2.4. Memory Scheduling on CPUs.

Memory scheduling algorithms improve system performance by reordering memory requests to deal with the different constraints and behaviors of DRAM. The first-ready-first-come-first-serve (FR-FCFS) [317] algorithm attempts to schedule requests that result in row-buffer hits (first-ready), and otherwise prioritizes older requests (FCFS). FR-FCFS increases DRAM throughput, but it can cause fairness problems by under-servicing applications with low row-buffer locality. Ebrahimi et

al. [93] propose PAM, a memory scheduler that prioritizes critical threads in order to improve the performance of multithreaded applications. Ipek et al. propose a self-optimizing memory scheduling that improve system performance with reinforcement learning [360]. Mukundan and Martinez propose MORSE, a self-optimizing reconfigurable memory scheduler [255]. Lee et al. propose two prefetch aware memory scheduling designs [209, 212]. Stuecheli et al. [352] and Lee et al. [211] propose memory schedulers that are aware of writeback requests. Seshadri et al. [329] propose to simplify the implementation of row-locality-aware write back by exploiting the dirty-block index. Several application-aware memory scheduling algorithms [197, 198, 253, 260, 261, 353, 357] have been proposed to balance both performance and fairness. Parallelism-aware Batch Scheduling (PAR-BS) [261] batches requests based on their arrival times (older requests batched first). Within a batch, applications are ranked to preserve bank-level parallelism (BLP) within an application's requests. Kim et al. propose ATLAS [197], which prioritizes applications that have received the least memory service. As a result, applications with low memory intensities, which typically attain low memory service, are prioritized. However, applications with high memory intensities are deprioritized and hence slowed down significantly, resulting in unfairness. Kim et al. further propose TCM [198], which addresses the unfairness problem in ATLAS. TCM first clusters applications into low and high memory-intensity clusters based on their memory intensities. TCM always prioritizes applications in the low memory-intensity cluster, however, among the high memory-intensity applications it shuffles request priorities to prevent unfairness. Ghose et al. propose a memory scheduler that takes into account of the criticality of each load and prioritizes loads that are more critical to CPU performance [119]. Subramanian et al. propose MISE [357], which is a memory scheduler that estimates slowdowns of applications and prioritizes applications that are likely to be slow down the most. Subramanian et al. also propose BLISS [353, 355], which is a mechanism that separates applications into a group that interferes with other applications and another group that does not, and prioritizes the latter group to increase performance and fairness. Xiong et al. propose DMPS, a ranking based on latency sensitivity [392]. Liu et al. propose LAMS, a memory scheduler that prioritizes requests based on the latency of servicing each memory request [230].

3.2.5. Memory Scheduling on GPUs.

Since GPU applications are bandwidth intensive, often with streaming access patterns, a policy that maximizes the number of row-buffer hits is effective for GPUs to maximize overall throughput. As a result, FR-FCFS with a large request buffer tends to perform well for GPUs [41]. In view of this, previous work [397] designed mechanisms to reduce the complexity of row-hit first based (FR-FCFS) scheduling. Jeong et al. propose a QoS-aware memory scheduler that guarantees the performance of GPU applications by prioritizing Graphics applications over CPU applications until the system can guarantee a frame can be rendered within its deadline, and prioritize CPU applications afterward [171]. Jog et al. [177] propose CLAM, a memory scheduler that identifies critical memory requests and prioritizes them in the main memory.

Aside from CPU-GPU heterogeneous systems, Usui et at. propose SQUASH [371] and DASH [372], which are accelerator-aware memory controller designs that improve the performance of systems with CPU and hardware accelerators. Zhao et al. propose FIRM, a memory controller design that improves the performance of systems with persistent memory [402].

3.2.6. DRAM Designs

Aside from memory scheduling and memory partitioning techniques, previous works propose new DRAM designs that are capable of reducing memory latency in conventional DRAM [22, 23, 60, 62, 63, 64, 66, 139, 148, 153, 192, 199, 213, 214, 215, 216, 217, 235, 247, 281, 298, 324, 337, 346, 383, 404] as well as non-volatile memory [203, 206, 207, 208, 245, 246, 308, 311, 395]. Previous works on bulk data transfer [58, 64, 131, 132, 159, 173, 181, 233, 328, 331, 400, 403] and in-memory computation [18, 21, 24, 38, 53, 85, 102, 109, 115, 116, 120, 133, 151, 152, 183, 201, 238, 283, 289, 290, 306, 330, 332, 333, 350, 359, 399] can be used improve DRAM bandwidth. Techniques to reduce the overhead of DRAM refresh can be applied to improve the performance of GPU-based systems [16, 17, 39, 47, 195, 225, 228, 263, 282, 309, 374]. Data compression techniques can also be used on the main memory to increase the effective DRAM bandwidth [292, 293, 294, 295, 378]. These techniques can be used to mitigate the performance impact of memory interference and

improve the performance of GPU-based systems. They are orthogonal and can be combined with techniques proposed in this dissertation.

Previous works on data prefetching can also be used to mitigate high DRAM latency [25,40,57, 75, 78, 91, 94, 95, 140, 141, 154, 179, 180, 205, 209, 210, 212, 258, 259, 262, 266, 335, 349]. However, these techniques generally increase DRAM bandwidth, which lead to lower GPU performance.

3.2.7. Interconnect Contention Management

Aside from the shared cache and the shared off-chip main memory, on-chip interconnect is another shared resources on the GPU memory hierarchy. While this dissertation does not focus on the contention of shared on-chip interconnect, many previous works provide mechanisms to reduce contention of the shared on-chip interconnect. These include works on hierarchical on-chip network designs [34, 35, 82, 88, 126, 135, 137, 313, 314, 401], low cost router designs [2, 34, 35, 127, 196, 200, 256], bufferless interconnect designs [27, 42, 61, 99, 100, 101, 121, 144, 149, 202, 254, 279, 280, 344] and Quality-of-Service-aware interconnect designs [81, 83, 84, 103, 128, 129, 130, 250].

3.3. Background on Memory Management Unit and Address Translation Designs

Aside from the caches and the main memory, the memory management unit (MMU) is another important component in the memory hierarchy. The MMU provides address translation for applications running on the GPU. When multiple GPGPU applications are concurrently running, the MMU is also provides memory protection across different virtual address spaces that are concurrently using the GPU memory. This section first introduces previous works on concurrent GPGPU application. Then, we provide background on previous works on TLB designs that aids address translation.

3.3.1. Background on Concurrent Execution of GPGPU Applications

Concurrent Kernels and GPU Multiprogramming. The opportunity to improve utilization with concurrency is well-recognized but previous proposals [223, 284, 382], do not support memory protection. Adriaens et al. [4] observe the need for spatial sharing across protection domains but do not propose or evaluate a design. NVIDIA GRID [147] and AMD FirePro [9] support static partitioning of hardware to allow kernels from different VMs to run concurrently—partitions are determined at startup, causing fragmentation and under-utilization. The goal of our proposal, MASK, is a flexible dynamic partitioning of shared resources. NVIDIA's Multi Process Service (MPS) [275] allows multiple processes to launch kernels on the GPU: the service provides no memory protection or error containment. Xu et al [393] propose Warped-Slicer, which is a mechanism for multiple applications to spatially share a GPU core. Similar to MPS, Warped-Slicer provides no memory protection, and is not suitable for supporting multi-application in a multi-tenant cloud setting.

Preemption and Context Switching. Preemptive context switching is an active research area [118, 364, 382]. Current architectural support [226, 276] will likely improve in future GPUs. Preemption and spatial multiplexing are complementary to the goal of this dissertation, and exploring techniques to combine them is future work.

GPU Virtualization. Most current hypervisor-based full virtualization techniques for GPGPUs [188, 361, 368] must support a virtual device abstraction without dedicated hardware support for VDI found in GRID [147] and FirePro [9] . Key components missing from these proposals includes support for dynamic partitioning of hardware resources and efficient techniques for handling over-subscription. Performance overheads incurred by some of these designs argue strongly for hardware assists such as those we propose. By contrast, API-remoting solutions such as vmCUDA [381] and rCUDA [87] provide near native performance but require modifications to guest software and sacrifice both isolation and compatibility.

Other Methods to Enable Virtual Memory. Vesely et al. analyze support for virtual memory in heterogeneous systems [375], finding that the cost of address translation in GPUs is an order of

magnitude higher than in CPUs and that high latency address translations limit the GPU's latency hiding capability and hurts performance (an observation in-line with our own findings. We show additionally that thrashing due to interference further slows applications sharing the GPU. Our proposal, MASK, is capable not only of reducing interference between multiple applications, but of reducing the TLB miss rate in single-application scenarios as well. We expect that our techniques are applicable to CPU-GPU heterogeneous system.

Direct segments [46] and redundant memory mappings [184] reduce address translation overheads by mapping large contiguous virtual memory to contiguous physical address space which reduces address translation overheads by increasing the reach of TLB entries. These techniques are complementary to those in MASK, and may eventually become relevant in GPU settings as well.

Demand Paging in GPUs. Demand paging is an important functionality for memory virtualization that is challenging for GPUs [375]. Recent works [405], AMD's hUMA [12], as well as NVIDIA's PASCAL architecture [276, 405] support for demand paging in GPUs. As identified in MOSAIC, these techniques can be costly in GPU environment.

3.3.2. TLB Designs

GPU TLB Designs. Previous works have explored the design space for TLBs in heterogeneous systems with GPUs [73, 302, 303, 375], and the adaptation of x86-like TLBs to a heterogeneous CPU-GPU setting [303]. Key elements in these designs include probing the TLB after L1 coalescing to reduce the number of parallel TLB requests, shared concurrent page table walks, and translation caches to reduce main memory accesses. Our proposal, MASK, owes much to these designs, but we show empirically that contention patterns at the shared L2 layer require additional support to accommodate cross-context contention. Cong et al. propose a TLB design similar to our baseline GPU-MMU design [73]. However, this design utilizes the host (CPU) MMU to perform page walks, which is inapplicable in the context of multi-application GPUs. Pichai et al. [302] explore TLB design for heterogeneous CPU-GPU systems, and add TLB awareness to the exist-

ing CCWS GPU warp scheduler [318], which enables parallel TLB access on the L1 cache level, similar in concept to the Powers design [303]. Warp scheduling is orthogonal to our work: incorporating a TLB-aware CCWS warp scheduler to MASK could further improve performance.

CPU TLB Designs. Bhattacharjee et al. examine shared last-level TLB designs [51] as well as page walk cache designs [48], proposing a mechanism that can accelerate multithreaded applications by sharing translations between cores. However, these proposals are likely to be less effective for multiple concurrent GPGPU applications because translations are not shared between virtual address spaces. Barr et al. propose SpecTLB [45], which speculatively predicts address translations to avoid the TLB miss latency. Speculatively predicting address translation can be complicated and costly in GPU because there can be multiple concurrent TLB misses to many different TLB entries in the GPU.

Mechanisms to Support Multiple Page Sizes. TLB miss overheads can be reduced by accelerating page table walks [44, 48] or reducing their frequency [112]; by reducing the number of TLB misses (e.g. through prefetching [50, 182, 325], prediction [286], or structural change to the TLB [299, 300, 363] or TLB hierarchy [19, 20, 46, 49, 113, 184, 236, 348]). Multipage mapping techniques [299, 300, 363] map multiple pages with a single TLB entry, improving TLB reach by a small factor (e.g., to 8 or 16); much greater improvements to TLB reach are needed to deal with modern memory sizes. Direct segments [46, 113] extend standard paging with a large segment to map the majority of an address space to a contiguous physical memory region, but require application modifications and are limited to workloads able to a single large segment. Redundant memory mappings (RMM) [184] extend TLB reach by mapping *ranges* of virtually and physically contiguous pages in a range TLB.

A number of related works propose hardware support to recover and expose address space contiguity. GLUE [301] groups contiguous, aligned small page translations under a single speculative large page translation in the TLB. Speculative translations (similar to SpecTLB [45]) can be verified by off-critical-path page table walks, reducing effective page-table walk latency. GTSM [86] provides hardware support to leverage the address space contiguity of physical memory even when

pages have been retired due to bit errors. Were such features to become available, hardware mechanisms for preserving address space contiguity could reduce the overheads induced by proactive compaction, which is a feature we introduce in our proposal, Mosaic.

The policies and mechanisms used to implement transparent large page support in Mosaic are informed by a wealth of previous research on operating system support for large pages for CPUs. Navarro et al. [265] identify contiguity-awareness and fragmentation reduction as primary concerns for large page management, proposing reservation-based allocation and deferred promotion of base pages to large pages. These ideas are widely used in modern operating systems [367]. Ingens [204] eschews reservation-based allocation in favor of the utilization-based promotion based on a bit vector which tracks spatial and temporal utilization of base pages, implementing promotion and demotion asynchronously, rather than in a page fault handler. These basic ideas heavily inform Mosaic's design, which attempts to emulate these same policies in hardware. In contrast to Ingens, Mosaic can rely on dedicated hardware to provide access frequency and distribution, and need not infer it by sampling access bits whose granularity may be a poor fit for the page size.

Gorman et al. [122] propose a placement policy for an operating system's physical page allocator that mitigates fragmentation and promotes address space contiguity by grouping pages according to relocatability. Subsequent work [123] proposes a software-exposed interface for applications to explicitly request large pages like `libhugetlbfs` [224]. These ideas are complementary to our work. Mosaic can plausibly benefit from similar policies simplified to be hardware-implementable, and we leave that investigation as future work.

Chapter 4

Reducing Intra-application Interference with Memory Divergence Correction

Graphics Processing Units (GPUs) have enormous parallel processing power to leverage thread-level parallelism. GPU applications can be broken down into thousands of threads, allowing GPUs to use *fine-grained multithreading* [345, 365] to prevent GPU cores from stalling due to dependencies and long memory latencies. Ideally, there should always be available threads for GPU cores to continue execution, preventing stalls within the core. GPUs also take advantage of the *SIMD* (Single Instruction, Multiple Data) execution model [106]. The thousands of threads within a GPU application are clustered into *work groups* (or *thread blocks*), with each thread block consisting of multiple smaller bundles of threads that are run concurrently. Each such thread bundle is called a *wavefront* [11] or *warp* [226]. In each cycle, each GPU core executes a single warp. Each thread in a warp executes the same instruction (i.e., is at the same program counter). Combining SIMD execution with fine-grained multithreading allows a GPU to complete several hundreds of operations every cycle in the ideal case.

In the past, GPUs strictly executed graphics applications, which naturally exhibit large amounts of concurrency. In recent years, with tools such as CUDA [274] and OpenCL [194], programmers have been able to adapt non-graphics applications to GPUs, writing these applications to have

thousands of threads that can be run on a SIMD computation engine. Such adapted non-graphics programs are known as general-purpose GPU (GPGPU) applications. Prior work has demonstrated that many scientific and data analysis applications can be executed significantly faster when programmed to run on GPUs [56, 68, 145, 351].

While many GPGPU applications can tolerate a significant amount of memory latency due to their parallelism and the use of fine-grained multithreading, many previous works (e.g., [175, 176, 264, 378]) observe that GPU cores still stall for a significant fraction of time when running many other GPGPU applications. One significant source of these stalls is *memory divergence*, where the threads of a warp reach a memory instruction, and some of the threads' memory requests take longer to service than the requests from other threads [65, 243, 264]. Since all threads within a warp operate in lockstep due to the SIMD execution model, the warp cannot proceed to the next instruction until the *slowest* request within the warp completes, and *all* threads are ready to continue execution. Figures 4.1a and 4.1b show examples of memory divergence within a warp, which we will explain in more detail soon.

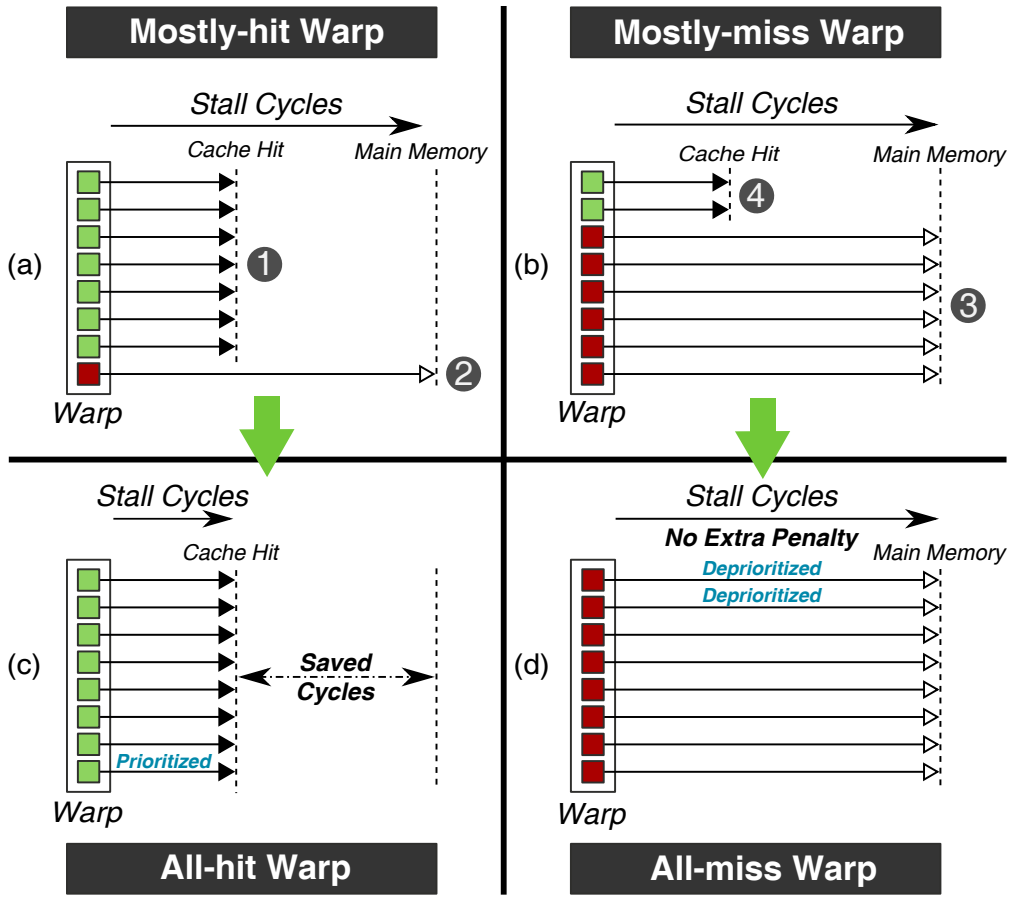


Figure 4.1. Memory divergence within a warp. (a) and (b) show the heterogeneity between *mostly-hit* and *mostly-miss* warps, respectively. (c) and (d) show the change in stall time from converting *mostly-hit warps into all-hit warps*, and *mostly-miss warps into all-miss warps*, respectively.

In this work, we make three new key observations about the memory divergence behavior of GPGPU warps:

Observation 1: There is *heterogeneity across warps* in the degree of memory divergence experienced by each warp at the shared L2 cache (i.e., the percentage of threads within a warp that miss in the cache varies widely). Figure 4.1 shows examples of two different *types of warps*, with eight threads each, that exhibit different degrees of memory divergence:

- Figure 4.1a shows a *mostly-hit warp*, where most of the warp's memory accesses hit in the cache (❶). However, a single access misses in the cache and must go to main memory (❷). As a result, the *entire warp* is stalled until the much longer cache miss completes.

- Figure 4.1b shows a *mostly-miss warp*, where most of the warp's memory requests miss in the cache (❸), resulting in many accesses to main memory. Even though some requests are cache hits (❹), these do not benefit the execution time of the warp.

Observation 2: *A warp tends to retain its memory divergence behavior* (e.g., whether or not it is mostly-hit or mostly-miss) *for long periods of execution*, and is thus predictable. As we show in Section 4.3, this predictability enables us to perform history-based warp divergence characterization.

Observation 3: Due to the amount of thread parallelism within a GPU, *a large number of memory requests can arrive at the L2 cache in a small window of execution time, leading to significant queuing delays*. Prior work observes high access latencies for the shared L2 cache within a GPU [340, 341, 385], but does not identify *why* these latencies are so high. We show that when a large number of requests arrive at the L2 cache, both the limited number of read/write ports and backpressure from cache bank conflicts force many of these requests to queue up for long periods of time. We observe that this queuing latency can sometimes add *hundreds* of cycles to the cache access latency, and that non-uniform queuing across the different cache banks exacerbates memory divergence.

Based on these three observations, we aim to devise a mechanism that has two major goals: (1) convert mostly-hit warps into *all-hit warps* (warps where *all* requests hit in the cache, as shown in Figure 4.1c), and (2) convert mostly-miss warps into *all-miss warps* (warps where *none* of the requests hit in the cache, as shown in Figure 4.1d). As we can see in Figure 4.1a, the stall time due to memory divergence for the mostly-hit warp can be eliminated by converting only the single cache miss (❷) into a hit. Doing so requires additional cache space. If we convert the two cache hits of the mostly-miss warp (Figure 4.1b, ❹) into cache misses, we can cede the cache space previously used by these hits to the mostly-hit warp, thus converting the mostly-hit warp into an all-hit warp. Though the mostly-miss warp is now an all-miss warp (Figure 4.1d), it incurs no extra stall penalty, as the warp was already waiting on the other six cache misses to complete. Additionally, now that it is an all-miss warp, we predict that its future memory requests will also

not be in the L2 cache, so we can simply have these requests *bypass the cache*. In doing so, the requests from the all-miss warp can completely avoid unnecessary L2 access and queuing delays. This decreases the total number of requests going to the L2 cache, thus reducing the queuing latencies for requests from mostly-hit and all-hit warps, as there is less contention.

We introduce *Memory Divergence Correction* (*MeDiC*), a GPU-specific mechanism that exploits *memory divergence heterogeneity* across warps at the shared cache and at main memory to improve the overall performance of GPGPU applications. MeDiC consists of three different components, which work together to achieve our goals of converting mostly-hit warps into all-hit warps and mostly-miss warps into all-miss warps: (1) a warp-type-aware *cache bypassing mechanism*, which prevents requests from mostly-miss and all-miss warps from accessing the shared L2 cache (Section 4.3.2); (2) a warp-type-aware *cache insertion policy*, which prioritizes requests from mostly-hit and all-hit warps to ensure that they all become cache hits (Section 4.3.3); and (3) a warp-type-aware *memory scheduling mechanism*, which prioritizes requests from mostly-hit warps that were not successfully converted to all-hit warps, in order to minimize the stall time due to divergence (Section 4.3.4). These three components are all driven by an online mechanism that can identify the expected memory divergence behavior of each warp (Section 4.3.1).

This dissertation makes the following contributions:

- We observe that the different warps within a GPGPU application exhibit heterogeneity in their memory divergence behavior at the shared L2 cache, and that some warps do not benefit from the few cache hits that they have. This memory divergence behavior tends to remain consistent throughout long periods of execution for a warp, allowing for fast, online warp divergence characterization and prediction.

- We identify a new performance bottleneck in GPGPU application execution that can contribute significantly to memory divergence: due to the very large number of memory requests issued by warps in GPGPU applications that contend at the shared L2 cache, many of these requests experience *high cache queuing latencies*.

- Based on our observations, we propose *Memory Divergence Correction*, a new mechanism that exploits the stable memory divergence behavior of warps to (1) improve the effectiveness of the cache by favoring warps that take the most advantage of the cache, (2) address the cache queuing problem, and (3) improve the effectiveness of the memory scheduler by favoring warps that benefit most from prioritization. We compare MeDiC to four different cache management mechanisms, and show that it improves performance by 21.8% and energy efficiency by 20.1% across a wide variety of GPGPU workloads compared to a a state-of-the-art GPU cache management mechanism [222].

4.1. Background

We first provide background on the architecture of a modern GPU, and then we discuss the bottlenecks that highly-multithreaded applications can face when executed on a GPU. These applications can be compiled using OpenCL [194] or CUDA [274], either of which converts a general purpose application into a GPGPU program that can execute on a GPU.

4.1.1. Baseline GPU Architecture

A typical GPU consists of several *shader cores* (sometimes called *streaming multiprocessors*, or SMs). In this work, we set the number of shader cores to 15, with 32 threads per warp in each core, corresponding to the NVIDIA GTX480 GPU based on the Fermi architecture [271]. The GPU we evaluate can issue up to 480 concurrent memory accesses per cycle [370]. Each core has its own private L1 data, texture, and constant caches, as well as a scratchpad memory [226, 271, 272]. In addition, the GPU also has several shared L2 cache slices and memory controllers. A *memory partition unit* combines a single L2 cache slice (which is banked) with a designated memory controller that connects to off-chip main memory. Figure 4.2 shows a simplified view of how the cores (or SMs), caches, and memory partitions are organized in our baseline GPU.

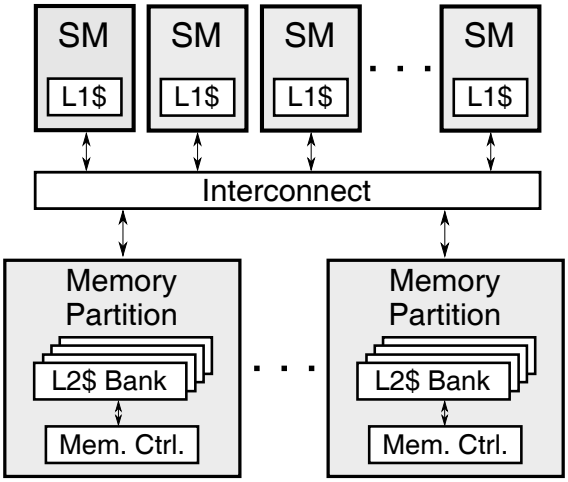


Figure 4.2. Overview of the baseline GPU architecture.

4.1.2. Bottlenecks in GPGPU Applications

Several previous works have analyzed the benefits and limitations of using a GPU for general purpose workloads (other than graphics purposes), including characterizing the impact of microarchitectural changes on applications [41] or developing performance models that break down performance bottlenecks in GPGPU applications [124, 150, 218, 231, 237, 338]. All of these works show benefits from using a throughput-oriented GPU. However, a significant number of applications are unable to fully utilize all of the available parallelism within the GPU, leading to periods of execution where no warps are available for execution [378].

When there are no available warps, the GPU cores stall, and the application stops making progress until a warp becomes available. Prior work has investigated two problems that can delay some warps from becoming available for execution: (1) *branch divergence*, which occurs when a branch in the same SIMD instruction resolves into multiple different paths [41, 110, 138, 264, 388], and (2) *memory divergence*, which occurs when the simultaneous memory requests from a single warp spend different amounts of time retrieving their associated data from memory [65, 243, 264]. In this work, we focus on the memory divergence problem; prior work on branch divergence is complementary to our work.

4.2. Motivation and Key Observations

We make three new key observations about memory divergence (at the shared L2 cache). First, we observe that the degree of memory divergence can differ across warps. This inter-warp heterogeneity affects how well each warp takes advantage of the shared cache. Second, we observe that a warp's memory divergence behavior tends to remain stable for long periods of execution, making it predictable. Third, we observe that requests to the shared cache experience long queuing delays due to the large amount of parallelism in GPGPU programs, which exacerbates the memory divergence problem and slows down GPU execution. Next, we describe each of these observations in detail and motivate our solutions.

4.2.1. Exploiting Heterogeneity Across Warps

We observe that different warps have different amounts of sensitivity to memory latency and cache utilization. We study the cache utilization of a warp by determining its *hit ratio*, the percentage of memory requests that hit in the cache when the warp issues a single memory instruction. As Figure 4.3 shows, the warps from each of our three representative GPGPU applications are distributed across all possible ranges of *hit ratio*, exhibiting significant heterogeneity. To better characterize warp behavior, we break the warps down into the five types shown in Figure 4.4 based on their hit ratios: *all-hit*, *mostly-hit*, *balanced*, *mostly-miss*, and *all-miss*.

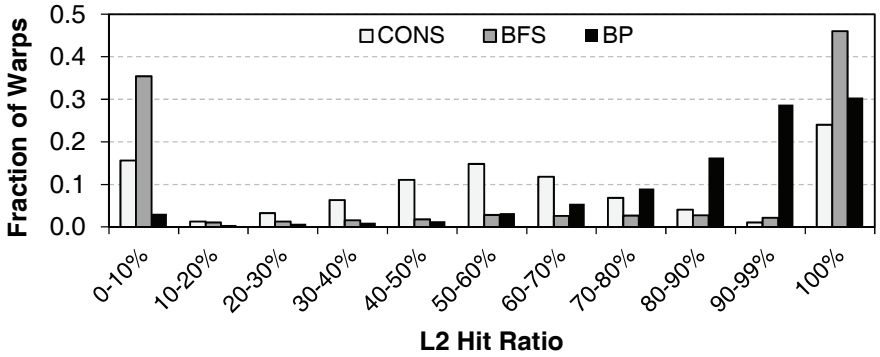


Figure 4.3. L2 cache hit ratio of different warps in three representative GPGPU applications (see Section 4.4 for methods).

Figure 4.4. Warp type categorization based on the shared cache hit ratios. Hit ratio values are empirically chosen.

This inter-warp heterogeneity in cache utilization provides new opportunities for performance improvement. We illustrate two such opportunities by walking through a simplified example, shown in Figure 4.5. Here, we have two warps, *A* and *B*, where *A* is a mostly-miss warp (with three of its four memory requests being L2 cache misses) and *B* is a mostly-hit warp with only a single L2 cache miss (request *B0*). Let us assume that warp *A* is scheduled first.

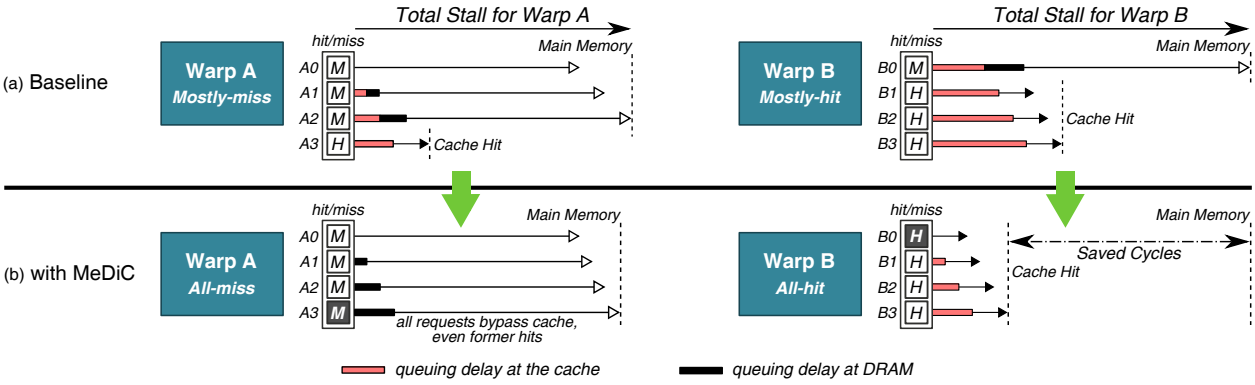


Figure 4.5. (a) Existing inter-warp heterogeneity, (b) exploiting the heterogeneity with MeDiC to improve performance.

As we can see in Figure 4.5a, the mostly-miss warp *A* does not benefit at all from the cache: even though one of its requests (*A3*) hits in the cache, warp *A* cannot continue executing until *all* of its memory requests are serviced. As the figure shows, using the cache to speed up only request *A3* has no material impact on warp *A*'s stall time. In addition, while requests *A1* and *A2* do not hit in the cache, they still incur a queuing latency at the cache while they wait to be looked up in the

cache tag array.

On the other hand, the mostly-hit warp *B* can be penalized significantly. First, since warp *B* is scheduled after the mostly-miss warp *A*, all four of warp *B*'s requests incur a large L2 queuing delay, *even though the cache was not useful to speed up warp* A. On top of this unproductive delay, since request *B0* misses in the cache, it holds up execution of the entire warp while it gets serviced by main memory. The overall effect is that despite having *many more* cache hits (and thus much better cache utility) than warp *A*, warp *B* ends up stalling for as long as or even longer than the mostly-miss warp *A* stalled for.

To remedy this problem, we set two goals (Figure 4.5b):

1) Convert the mostly-hit warp B *into an all-hit warp.* By converting *B0* into a hit, warp *B* no longer has to stall on any memory misses, which enables the warp to become ready to execute much earlier. This requires a little additional space in the cache to store the data for *B0*.

2) Convert the mostly-miss warp A *into an all-miss warp.* Since a single cache hit is of no effect to warp *A*'s execution, we convert *A0* into a cache miss. This frees up the cache space *A0* was using, and thus creates cache space for storing *B0*. In addition, warp *A*'s requests can now skip accessing the cache and go straight to main memory, which has two benefits: *A0–A2* complete faster because they no longer experience the cache queuing delay that they incurred in Figure 4.5a, and *B0–B3* also complete faster because they must queue behind a smaller number of cache requests. Thus, bypassing the cache for warp *A*'s request allows *both* warps to stall for less time, improving GPU core utilization.

To realize these benefits, we propose to (1) develop a mechanism that can identify mostly-hit and mostly-miss warps; (2) design a mechanism that allows mostly-miss warps to yield their ineffective cache space to mostly-hit warps, similar to how the mostly-miss warp *A* in Figure 4.5a turns into an all-miss warp in Figure 4.5b, so that warps such as the mostly-hit warp *B* can become all-hit warps; (3) design a mechanism that bypasses the cache for requests from mostly-miss and all-miss warps such as warp *A*, to decrease warp stall time and reduce lengthy cache queuing latencies; and (4) prioritize requests from mostly-hit warps across the memory hierarchy, at both

the shared L2 cache and at the memory controller, to minimize their stall time as much as possible, similar to how the mostly-hit warp *B* in Figure 4.5a turns into an all-hit warp in Figure 4.5b.

A key challenge is how to group warps into different warp types. In this work, we observe that warps tend to exhibit stable cache hit behavior over long periods of execution. A warp consists of several threads that repeatedly loop over the same instruction sequences. This secs/medic/results in similar hit/miss behavior at the cache level across different instances of the same warp. As a result, a warp measured to have a particular hit ratio is likely to maintain a similar hit ratio throughout a lengthy phase of execution. We observe that most CUDA applications exhibit this trend.

Figure 4.6 shows the hit ratio over a duration of one million cycles, for six randomly selected warps from our CUDA applications. We also plot horizontal lines to illustrate the hit ratio cutoffs that we set in Figure 4.4 for our mostly-hit ($\geq$70%) and mostly-miss ($\leq$20%) warp types. Warps 1, 3, and 6 spend the majority of their time with high hit ratios, and are classified as mostly-hit warps. Warps 1 and 3 do, however, exhibit some long-term (i.e., 100k+ cycles) shifts to the balanced warp type. Warps 2 and 5 spend a long time as mostly-miss warps, though they both experience a single long-term shift into balanced warp behavior. As we can see, *warps tend to remain in the same warp type* at least for hundreds of thousands of cycles.

As a result of this relatively stable behavior, our mechanism, MeDiC (described in detail in Section 4.3), samples the hit ratio of each warp and uses this data for warp characterization. To account for the long-term hit ratio shifts, MeDiC resamples the hit ratio every 100k cycles.

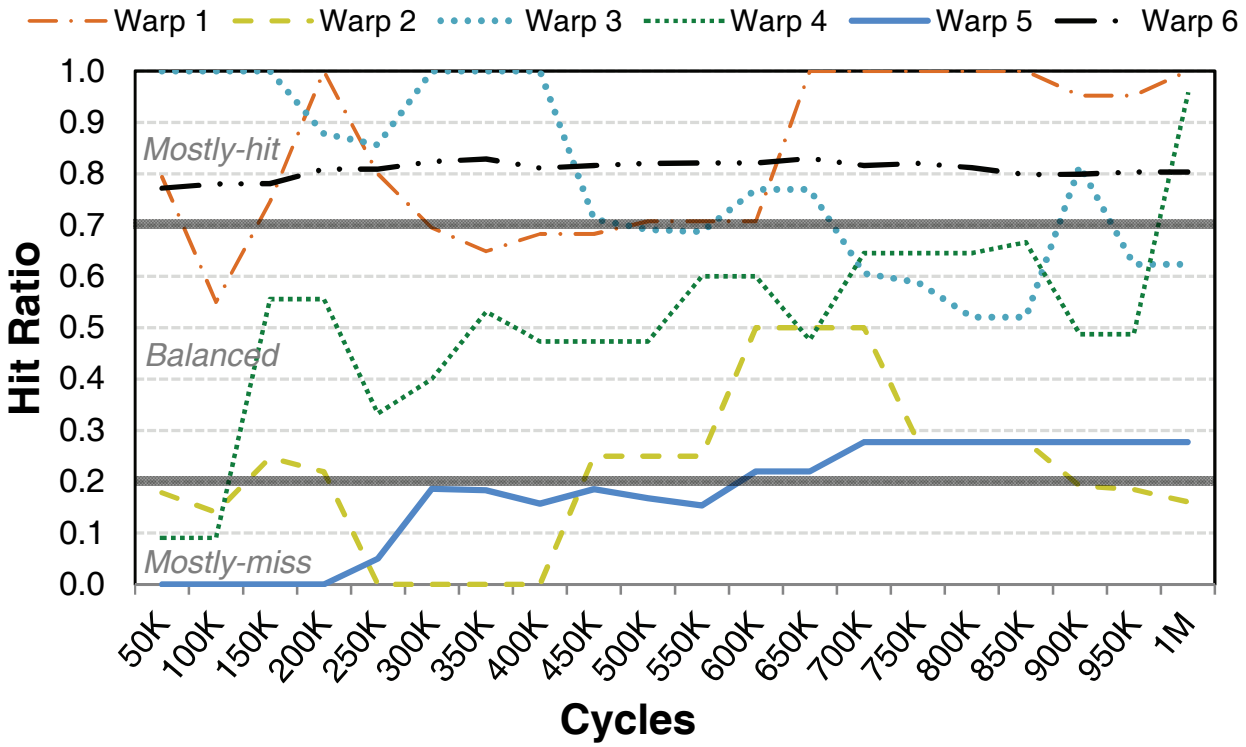


Figure 4.6. Hit ratio of randomly selected warps over time.

4.2.2. Reducing the Effects of L2 Queuing Latency

Unlike CPU applications, GPGPU applications can issue as many as hundreds of memory instructions per cycle. All of these memory requests can arrive concurrently at the L2 cache, which is the first shared level of the memory hierarchy, creating a bottleneck. Previous works [41, 340, 341, 385] point out that the latency for accessing the L2 cache can take hundreds of cycles, even though the nominal cache lookup latency is significantly lower (only tens of cycles). While they identify this disparity, these earlier efforts do not identify or analyze the source of these long delays.

We make a new observation that identifies an important source of the long L2 cache access delays in GPGPU systems. L2 bank conflicts can cause queuing delay, which can differ from one bank to another and lead to the disparity of cache access latencies across different banks. As Figure 4.7a shows, even if every cache access within a warp hits in the L2 cache, each access can incur a different cache latency due to non-uniform queuing, and the warp has to stall until

the *slowest* cache access retrieves its data (i.e., memory divergence can occur). For each set of simultaneous requests issued by an all-hit warp, we define its *inter-bank divergence penalty* to be the difference between the fastest cache hit and the slowest cache hit, as depicted in Figure 4.7a.

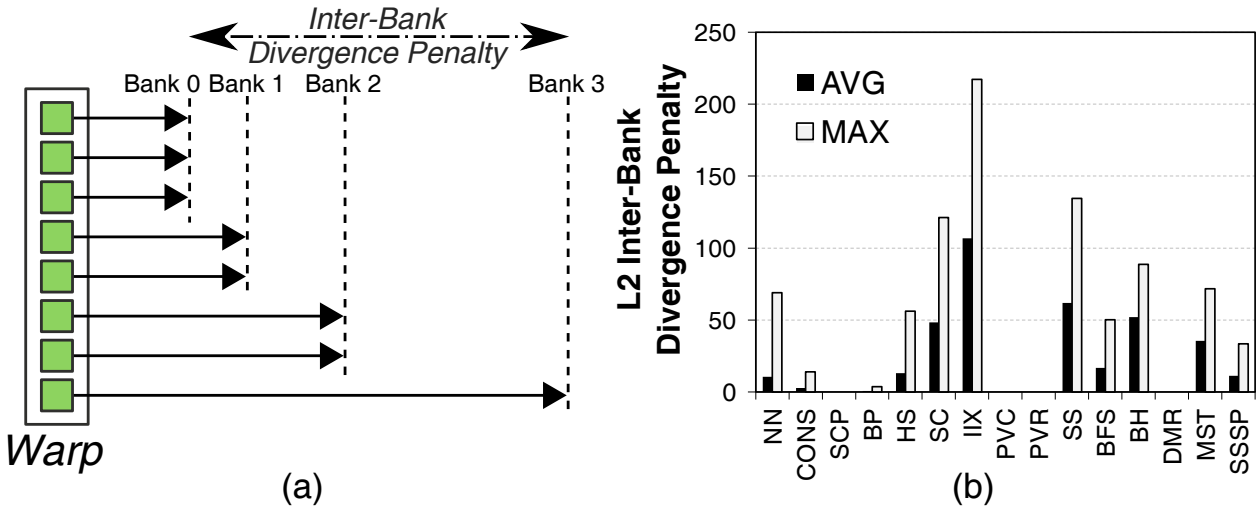


Figure 4.7. Effect of bank queuing latency divergence in the L2 cache: (a) example of the impact on stall time of skewed queuing latencies, (b) inter-bank divergence penalty due to skewed queuing for all-hit warps, in cycles.

In order to confirm this behavior, we modify GPGPU-Sim [41] to accurately model L2 bank conflicts and queuing delays (see Section 4.4 for details). We then measure the average and maximum inter-bank divergence penalty observed *only for all-hit warps* in our different CUDA applications, shown in Figure 4.7b. We find that *on average*, an all-hit warp has to stall for an additional 24.0 cycles because some of its requests go to cache banks with high access contention.

To quantify the magnitude of queue contention, we analyze the queuing delays for a two-bank L2 cache where the tag lookup latency is set to one cycle. We find that even with such a small cache lookup latency, a significant number of requests experience tens, if not hundreds, of cycles of queuing delay. Figure 4.8 shows the distribution of these delays for BFS [56], across all of its individual L2 cache requests. BFS contains one compute-intensive kernel and two memory-intensive kernels. We observe that requests generated by the compute-intensive kernel do not incur high queuing latencies, while requests from the memory-intensive kernels suffer from significant queuing delays. On average, across all three kernels, cache requests spend 34.8 cycles in the

queue waiting to be serviced, which is quite high considering the idealized one-cycle cache lookup latency.

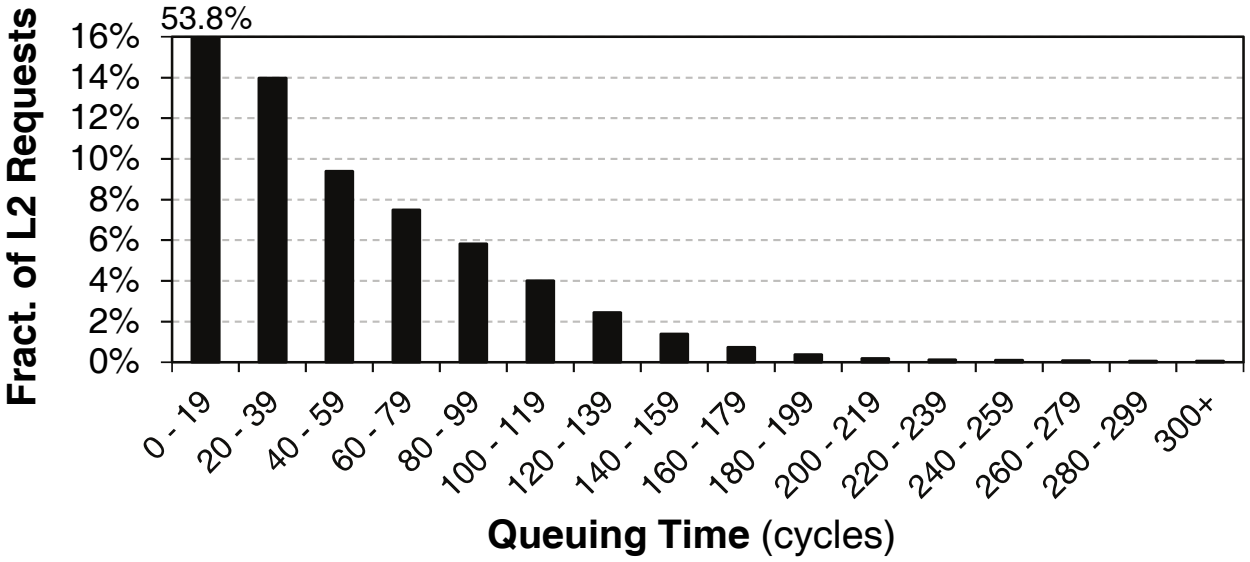


Figure 4.8. Distribution of per-request queuing latencies for L2 cache requests from BFS.

One naive solution to the L2 cache queuing problem is to increase the number of banks, without reducing the number of physical ports per bank and without increasing the size of the shared cache. However, as shown in Figure 4.9, the average performance improvement from doubling the number of banks to 24 (i.e., 4 banks per memory partition) is less than 4%, while the improvement from quadrupling the banks is less than 6%. There are two key reasons for this minimal performance gain. First, while more cache banks can help to distribute the queued requests, these extra banks do not change the memory divergence behavior of the warp (i.e., the warp hit ratios remain unchanged). Second, non-uniform bank access patterns still remain, causing cache requests to queue up unevenly at a few banks.[1]

4.2.3. Our Goal

Our goal of MeDiC is to improve cache utilization and reduce cache queuing latency by taking advantage of heterogeneity between different types of warps. To this end, we create a mechanism that (1) tries to eliminate mostly-hit and mostly-miss warps by converting as many of them as

[1]Similar problems have been observed for bank conflicts in main memory [199, 312].

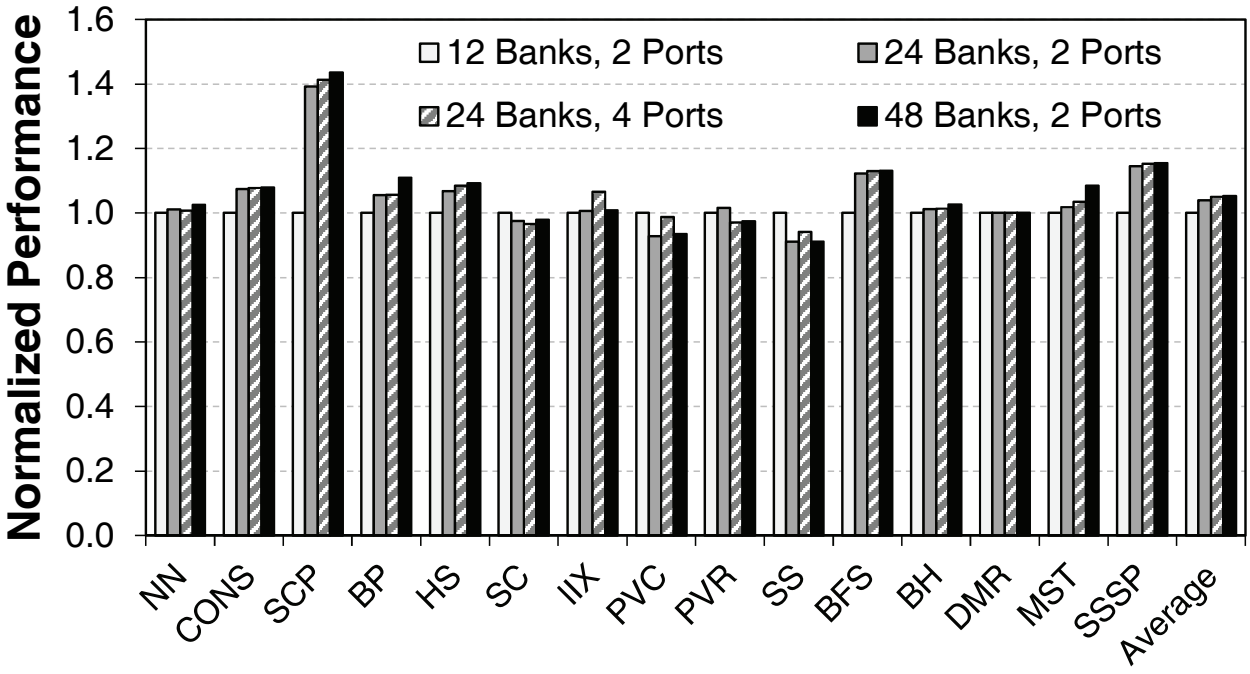


Figure 4.9. Performance of GPGPU applications with different number of banks and ports per bank, normalized to a 12-bank cache with 2 ports per bank.

possible to all-hit and all-miss warps, respectively; (2) reduces the queuing delay at the L2 cache by bypassing requests from mostly-miss and all-miss warps, such that each L2 cache hit experiences a much lower overall L2 cache latency; and (3) prioritizes mostly-hit warps in the memory scheduler to minimize the amount of time they stall due to a cache miss.

4.3. MeDiC: Memory Divergence Correction

In this section, we introduce Memory Divergence Correction (MeDiC), a set of techniques that take advantage of the memory divergence heterogeneity across warps, as discussed in Section 4.2. These techniques work independently of each other, but act synergistically to provide a substantial performance improvement. In Section 4.3.1, we propose a mechanism that identifies and groups warps into different warp types based on their degree of memory divergence, as shown in Figure 4.4.

As depicted in Figure 4.10, MeDiC uses ❶ warp type identification to drive three different components: ❷ a *warp-type-aware cache bypass mechanism* (Section 4.3.2), which bypasses re-

quests from all-miss and mostly-miss warps to reduce the L2 queuing delay; ❸ a *warp-type-aware cache insertion policy* (Section 4.3.3), which works to keep cache lines from mostly-hit warps while demoting lines from mostly-miss warps; and ❹ a *warp-type-aware memory scheduler* (Section 4.3.4), which prioritizes DRAM requests from mostly-hit warps as they are highly latency sensitive. We analyze the hardware cost of MeDiC in Section 4.5.5.

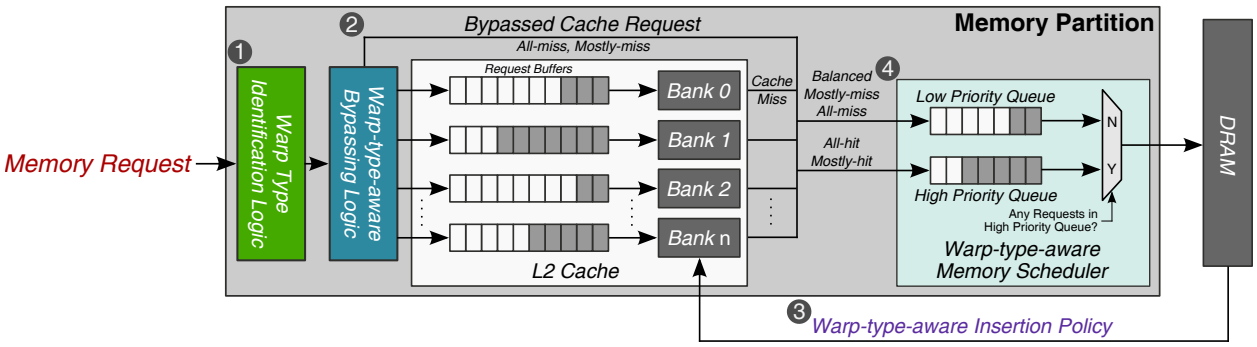


Figure 4.10. Overview of MeDiC: ❶ warp type identification logic, ❷ warp-type-aware cache bypassing, ❸ warp-type-aware cache insertion policy, ❹ warp-type-aware memory scheduler.

4.3.1. Warp Type Identification

In order to take advantage of the memory divergence heterogeneity across warps, we must first add hardware that can identify the divergence behavior of each warp. The key idea is to periodically sample the hit ratio of a warp, and to classify the warp's divergence behavior as one of the five types in Figure 4.4 based on the observed hit ratio (see Section 4.2.1). This information can then be used to drive the warp-type-aware components of MeDiC. In general, warps tend to retain the same memory divergence behavior for long periods of execution. However, as we observed in Section 4.2.1, there can be some long-term shifts in warp divergence behavior, requiring periodic resampling of the hit ratio to potentially adjust the warp type.

Warp type identification through hit ratio sampling requires hardware within the cache to periodically count the number of hits and misses each warp incurs. We append two counters to the metadata stored for each warp, which represent the total number of cache hits and cache accesses for the warp. We reset these counters periodically, and set the bypass logic to operate in a profiling

phase for each warp after this reset.[2] During profiling, which lasts for the first 30 cache accesses of each warp, the bypass logic (which we explain in Section 4.3.2) does not make any cache bypassing decisions, to allow the counters to accurately characterize the current memory divergence behavior of the warp. At the end of profiling, the warp type is determined and stored in the metadata.

4.3.2. Warp-type-aware Shared Cache Bypassing

Once the warp type is known and a warp generates a request to the L2 cache, our mechanism first decides whether to bypass the cache based on the warp type. The key idea behind *warp-type-aware cache bypassing*, as discussed in Section 4.2.1, is to convert mostly-miss warps into all-miss warps, as they do not benefit greatly from the few cache hits that they get. By bypassing these requests, we achieve three benefits: (1) bypassed requests can avoid L2 queuing latencies entirely, (2) other requests that do hit in the L2 cache experience shorter queuing delays due to the reduced contention, and (3) space is created in the L2 cache for mostly-hit warps.

The cache bypassing logic must make a simple decision: if an incoming memory request was generated by a mostly-miss or all-miss warp, the request is bypassed directly to DRAM. This is determined by reading the warp type stored in the warp metadata from the warp type identification mechanism. A simple 2-bit demultiplexer can be used to determine whether a request is sent to the L2 bank arbiter, or directly to the DRAM request queue.

Dynamically Tuning the Cache Bypassing Rate. While cache bypassing alleviates queuing pressure at the L2 cache banks, it can have a negative impact on other portions of the memory partition. For example, bypassed requests that were originally cache hits now consume extra off-chip memory bandwidth, and can increase queuing delays at the DRAM queue. If we lower the number of bypassed requests (i.e., reduce the number of warps classified as mostly-miss), we can reduce DRAM utilization. After examining a random selection of kernels from three applications (BFS, BP, and CONS), we find that the ideal number of warps classified as mostly-miss differs for each kernel. Therefore, we add a mechanism that *dynamically* tunes the hit ratio boundary between

[2] In this work, we reset the hit ratio every 100k cycles for each warp.

mostly-miss warps and balanced warps (nominally set at 20%; see Figure 4.4). If the cache miss rate increases significantly, the hit ratio boundary is lowered.[3]

Cache Write Policy. Recent GPUs support multiple options for the L2 cache write policy [271]. In this work, we assume that the L2 cache is write-through [340], so our bypassing logic can always assume that DRAM contains an up-to-date copy of the data. For write-back caches, previously-proposed mechanisms [134, 242, 339] can be used in conjunction with our bypassing technique to ensure that bypassed requests get the correct data. For correctness, fences and atomic instructions from bypassed warps still access the L2 for cache lookup, but are not allowed to store data in the cache.

4.3.3. Warp-type-aware Cache Insertion Policy

Our cache bypassing mechanism frees up space within the L2 cache, which we want to use for the cache misses from mostly-hit warps (to convert these memory requests into cache hits). However, even with the new bypassing mechanism, other warps (e.g., balanced, mostly-miss) still insert some data into the cache. In order to aid the conversion of mostly-hit warps into all-hit warps, we develop a *warp-type-aware cache insertion policy*, whose key idea is to ensure that for a given cache set, data from mostly-miss warps are evicted first, while data from mostly-hit warps and all-hit warps are evicted last.

To ensure that a cache block from a mostly-hit warp stays in the cache for as long as possible, we insert the block closer to the MRU position. A cache block requested by a mostly-miss warp is inserted closer to the LRU position, making it more likely to be evicted. To track the status of these cache blocks, we add two bits of metadata to each cache block, indicating the warp type.[4] These bits are then appended to the replacement policy bits. As a result, a cache block from a mostly-miss warp is more likely to get evicted than a block from a balanced warp. Similarly, a cache block from a balanced warp is more likely to be evicted than a block from a mostly-hit or

[3] In our evaluation, we reduce the threshold value between mostly-miss warps and balanced warps by 5% for every 5% increase in cache miss rate.

[4] Note that cache blocks from the all-miss category share the same 2-bit value as the mostly-miss category because they always get bypassed (see Section 4.3.2).

all-hit warp.

4.3.4. Warp-type-aware Memory Scheduler

Our cache bypassing mechanism and cache insertion policy work to increase the likelihood that *all* requests from a mostly-hit warp become cache hits, converting the warp into an all-hit warp. However, due to cache conflicts, or due to poor locality, there may still be cases when a mostly-hit warp cannot be fully converted into an all-hit warp, and is therefore unable to avoid stalling due to memory divergence as at least one of its requests has to go to DRAM. In such a case, we want to minimize the amount of time that this warp stalls. To this end, we propose a *warp-type-aware memory scheduler* that prioritizes the occasional DRAM request from a mostly-hit warp.

The design of our memory scheduler is very simple. Each memory request is tagged with a single bit, which is set if the memory request comes from a mostly-hit warp (or an all-hit warp, in case the warp was mischaracterized). We modify the request queue at the memory controller to contain two different queues (❹ in Figure 4.10), where a *high-priority queue* contains all requests that have their mostly-hit bit set to one. The *low-priority queue* contains all other requests, whose mostly-hit bits are set to zero. Each queue uses FR-FCFS [317, 406] as the scheduling policy; however, the scheduler always selects requests from the high priority queue over requests in the low priority queue.[5]

4.4. Methodology

We model our mechanism using GPGPU-Sim 3.2.1 [41]. Table 7.1 shows the configuration of the GPU. We modified GPGPU-Sim to accurately model cache bank conflicts, and added the cache bypassing, cache insertion, and memory scheduling mechanisms needed to support MeDiC. We use GPUWattch [219] to evaluate power consumption.

Modeling L2 Bank Conflicts. In order to analyze the detailed caching behavior of applications

[5]Using two queues ensures that high-priority requests are not blocked by low-priority requests even when the low-priority queue is full. Two-queue priority also uses simpler logic design than comparator-based priority [353, 354].

System Overview	15 cores, 6 memory partitions
Shader Core Config.	1400 MHz, 9-stage pipeline, GTO scheduler [318]
Private L1 Cache	16KB, 4-way associative, LRU, L1 misses are coalesced before ccessing L2, 1 cycle latency
Shared L2 Cache	768KB total, 16-way associative, LRU, 2 cache banks 2 interconnect ports per memory partition, 10 cycle latency
DRAM	GDDR5 1674 MHz, 6 channels (one per memory partition) FR-FCFS scheduler [317, 406] 8 banks per rank, burst length 8

Table 4.1. Configuration of the simulated system.

in modern GPGPU architectures, we modified GPGPU-Sim to accurately model banked caches.[6] Within each memory partition, we divide the shared L2 cache into two banks. When a memory request misses in the L1 cache, it is sent to the memory partition through the shared interconnect. However, it can only be sent if there is a free port available at the memory partition (we dual-port each memory partition). Once a request arrives at the port, a unified bank arbiter dispatches the request to the request queue for the appropriate cache bank (which is determined statically using some of the memory address bits). If the bank request queue is full, the request remains at the incoming port until the queue is freed up. Traveling through the port and arbiter consumes an extra cycle per request. In order to prevent a bias towards any one port or any one cache bank, the simulator rotates which port and which bank are first examined every cycle.

When a request misses in the L2 cache, it is sent to the DRAM request queue, which is shared across all L2 banks as previously implemented in GPGPU-Sim. When a request returns from DRAM, it is inserted into one of the per-bank DRAM-to-L2 queues. Requests returning from the L2 cache to the L1 cache go through a unified memory-partition-to-interconnect queue (where round-robin priority is used to insert requests from different banks into the queue).

GPGPU Applications. We evaluate our system across multiple GPGPU applications from the CUDA SDK [270], Rodinia [68], MARS [145], and Lonestar [56] benchmark suites.[7] These

[6] We validate that the performance values reported for our applications before and after our modifications to GPGPU-Sim are equivalent.

[7] We use default tuning parameters for all applications.

#	Application	AH	MH	BL	MM	AM
1	Nearest Neighbor (NN) [270]	19%	**79%**	1%	0.9%	0.1%
2	Convolution Separable (CONS) [270]	9%	1%	**82%**	1%	7%
3	Scalar Product (SCP) [270]	0.1%	0.1%	0.1%	0.7%	**99%**
4	Back Propagation (BP) [68]	10%	27%	**48%**	6%	9%
5	Hotspot (HS) [68]	1%	29%	**69%**	0.5%	0.5%
6	Streamcluster (SC) [68]	6%	0.2%	0.5%	0.3%	**93%**
7	Inverted Index (IIX) [145]	**71%**	5%	8%	1%	15%
8	Page View Count (PVC) [145]	4%	1%	**42%**	20%	33%
9	Page View Rank (PVR) [145]	18%	3%	28%	4%	**47%**
10	Similarity Score (SS) [145]	**67%**	1%	11%	1%	20%
11	Breadth-First Search (BFS) [56]	**40%**	1%	20%	13%	26%
12	Barnes-Hut N-body Simulation (BH) [56]	**84%**	0%	0%	1%	15%
13	Delaunay Mesh Refinement (DMR) [56]	**81%**	3%	3%	1%	12%
14	Minimum Spanning Tree (MST) [56]	**53%**	12%	18%	2%	15%
15	Survey Propagation (SP) [56]	**41%**	1%	20%	14%	24%

Table 4.2. Evaluated GPGPU applications and the characteristics of their warps.

applications are listed in Table 4.2, along with the breakdown of warp characterization. The dominant warp type for each application is marked in *bold* (AH: all-hit, MH: mostly-hit, BL: balanced, MM: mostly-miss, AM: all-miss; see Figure 4.4). We simulate 500 million instructions for each kernel of our application, though some kernels complete before reaching this instruction count.

Comparisons. In addition to the baseline secs/medic/results, we compare each individual component of MeDiC with state-of-the-art policies. We compare our bypassing mechanism with three different cache management policies. First, we compare to PCAL [222], a token-based cache management mechanism. PCAL limits the number of threads that get to access the cache by using tokens. If a cache request is a miss, it causes a replacement only if the warp has a token. PCAL, as modeled in this work, first grants tokens to the warp that recently used the cache, then grants any remaining tokens to warps that access the cache in order of their arrival. Unlike the original proposal [222], which applies PCAL to the L1 caches, we apply PCAL to the shared L2 cache. We sweep the number of tokens per epoch and use the configuration that gives the best average performance. Second, we compare MeDiC against a random bypassing policy (**Rand**), where a percentage of randomly-chosen warps bypass the cache every 100k cycles. For every workload,

we statically configure the percentage of warps that bypass the cache such that Rand yields the best performance. This comparison point is designed to show the value of warp type information in bypassing decisions. Third, we compare to a program counter (PC) based bypassing policy (**PC-Byp**). This mechanism bypasses requests from *static instructions* that mostly miss (as opposed to requests from mostly-miss *warps*). This comparison point is designed to distinguish the value of tracking hit ratios at the warp level instead of at the instruction level.

We compare our memory scheduling mechanism with the baseline first-ready, first-come first-serve (FR-FCFS) memory scheduler [317, 406], which is reported to provide good performance on GPU and GPGPU workloads [33, 65, 397]. We compare our cache insertion with the Evicted-Address Filter [334], a state-of-the-art CPU cache insertion policy.

Evaluation Metrics. We report performance secs/medic/results using the harmonic average of the IPC speedup (over the baseline GPU) of each kernel of each application.[8] Harmonic speedup was shown to reflect the average normalized execution time in multiprogrammed workloads [97]. We calculate energy efficiency for each workload by dividing the IPC by the energy consumed.

4.5. Evaluation

4.5.1. Performance Improvement of MeDiC

Figure 4.11 shows the performance of MeDiC compared to the various state-of-the-art mechanisms (EAF [334], PCAL [222], Rand, PC-Byp) from Section 4.4,[9] as well as the performance of each individual component in MeDiC.

Baseline shows the performance of the unmodified GPU using FR-FCFS as the memory scheduler [317, 406]. **EAF** shows the performance of the Evicted-Address Filter [334]. **WIP** shows the performance of our warp-type-aware insertion policy by itself. **WMS** shows the performance of

[8] We confirm that for each application, all kernels have similar speedup values, and that aside from SS and PVC, there are no outliers (i.e., no kernel has a much higher speedup than the other kernels). To verify that harmonic speedup is not swayed greatly by these few outliers, we recompute it for SS and PVC *without* these outliers, and find that the outlier-free speedup is within 1% of the harmonic speedup we report in the dissertation.

[9] We tune the configuration of each of these previously-proposed mechanisms such that those mechanisms achieve the highest performance secs/medic/results.

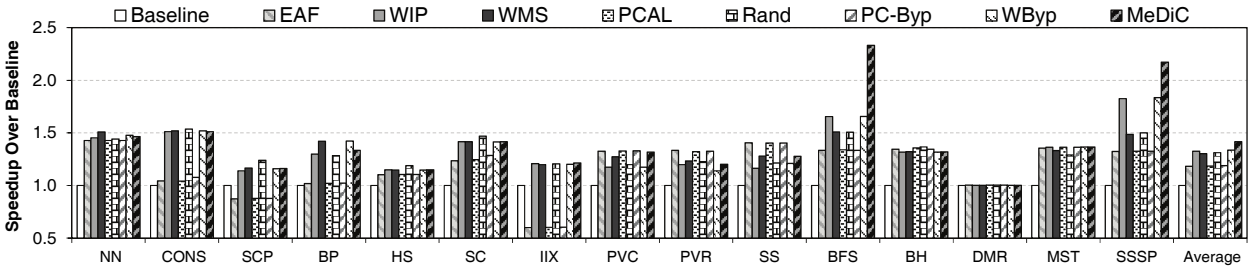


Figure 4.11. Performance of MeDiC.

our warp-type-aware memory scheduling policy by itself. **PCAL** shows the performance of the PCAL bypassing mechanism proposed by Li et al. [222]. **Rand** shows the performance of a cache bypassing mechanism that performs bypassing decisions randomly on a fixed percentage of warps. **PC-Byp** shows the performance of the bypassing mechanism that uses the PC as the criterion for bypassing instead of the warp-type. **WByp** shows the performance of our warp-type-aware bypassing policy by itself.

From these secs/medic/results, we draw the following conclusions:

- Each component of MeDiC individually provides significant performance improvement: WIP (32.5%), WMS (30.2%), and WByp (33.6%). MeDiC, which combines all three mechanisms, provides a 41.5% performance improvement over Baseline, on average. MeDiC matches or outperforms its individual components for all benchmarks except BP, where MeDiC has a higher L2 miss rate and lower row buffer locality than WMS and WByp.

- WIP outperforms EAF [334] by 12.2%. We observe that the key benefit of WIP is that cache blocks from mostly-miss warps are much more likely to be evicted. In addition, WIP reduces the cache miss rate of several applications (see Section 4.5.3).

- WMS provides significant performance gains (30.2%) over Baseline, because the memory scheduler prioritizes requests from warps that have a high hit ratio, allowing these warps to become active much sooner than they do in Baseline.

- WByp provides an average 33.6% performance improvement over Baseline, because it is effective at reducing the L2 queuing latency. We show the change in queuing latency and

provide a more detailed analysis in Section 4.5.3.

- Compared to PCAL [222], WByp provides 12.8% better performance, and full MeDiC provides 21.8% better performance. We observe that while PCAL reduces the amount of cache thrashing, the reduction in thrashing does not directly translate into better performance. We observe that warps in the mostly-miss category sometimes have high reuse, and acquire tokens to access the cache. This causes less cache space to become available for mostly-hit warps, limiting how many of these warps become all-hit. However, when high-reuse warps that possess tokens are mainly in the mostly-hit category (PVC, PVR, SS, and BH), we find that PCAL performs better than WByp.

- Compared to Rand,[10] MeDiC performs 6.8% better, because MeDiC is able to make bypassing decisions that do not increase the miss rate significantly. This leads to lower off-chip bandwidth usage under MeDiC than under Rand. Rand increases the cache miss rate by 10% for the kernels of several applications (BP, PVC, PVR, BFS, and MST). We observe that in many cases, MeDiC improves the performance of applications that tend to generate a large number of memory requests, and thus experience substantial queuing latencies. We further analyze the effect of MeDiC on queuing delay in Section 4.5.3.

- Compared to PC-Byp, MeDiC performs 12.4% better. We observe that the overhead of tracking the PC becomes significant, and that thrashing occurs as two PCs can hash to the same index, leading to inaccuracies in the bypassing decisions.

We conclude that each component of MeDiC, and the full MeDiC framework, are effective. Note that each component of MeDiC addresses the same problem (i.e., memory divergence of threads within a warp) using different techniques on different parts of the memory hierarchy. For the majority of workloads, one optimization is enough. However, we see that for certain high-intensity workloads (BFS and SSSP), the congestion is so high that we need to attack divergence

[10]Note that our evaluation uses an ideal random bypassing mechanism, where we manually select the best individual percentage of requests to bypass the cache for each workload. As a result, the performance shown for Rand is better than can be practically realized.

on multiple fronts. Thus, MeDiC provides better average performance than all of its individual components, especially for such memory-intensive workloads.

4.5.2. Energy Efficiency of MeDiC

MeDiC provides significant GPU energy efficiency improvements, as shown in Figure 4.12. All three components of MeDiC, as well as the full MeDiC framework, are more energy efficient than all of the other works we compare against. MeDiC is 53.5% more energy efficient than Baseline. WIP itself is 19.3% more energy efficient than EAF. WMS is 45.2% more energy efficient than Baseline, which uses an FR-FCFS memory scheduler [317, 406]. WByp and MeDiC are more energy efficient than all of the other evaluated bypassing mechanisms, with 8.3% and 20.1% more efficiency than PCAL [222], respectively.

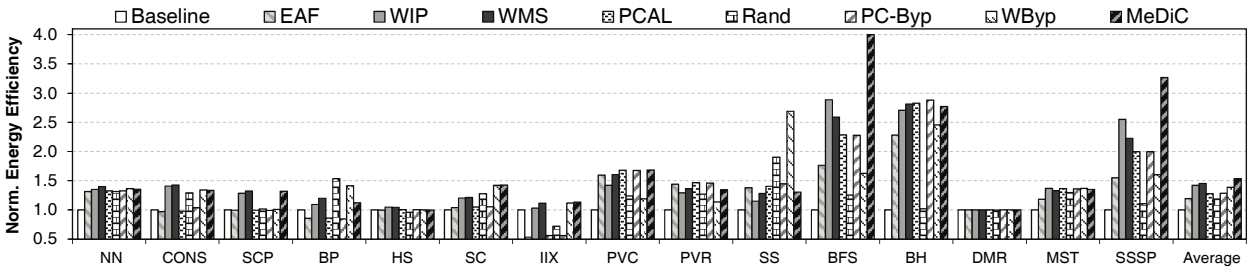


Figure 4.12. Energy efficiency of MeDiC.

For all of our applications, the energy efficiency of MeDiC is better than or equal to Baseline, because even though our bypassing logic sometimes increases energy consumption by sending more memory requests to DRAM, the resulting performance improvement outweighs this additional energy. We also observe that our insertion policy reduces the L2 cache miss rate, allowing MeDiC to be even more energy efficient by not wasting energy on cache lookups for requests of all-miss warps.

4.5.3. Analysis of Benefits

Impact of MeDiC on Cache Miss Rate. One possible downside of cache bypassing is that the bypassed requests can introduce extra cache misses. Figure 4.13 shows the cache miss rate for

Baseline, Rand, WIP, and MeDiC.

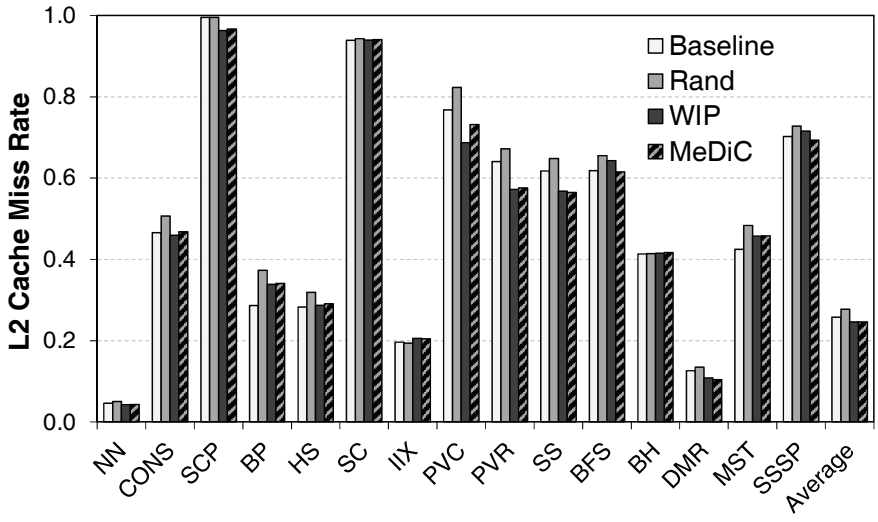


Figure 4.13. L2 Cache miss rate of MeDiC.

Unlike Rand, MeDiC does not increase the cache miss rate over Baseline for most of our applications. The key factor behind this is WIP, the insertion policy in MeDiC. We observe that WIP on its own provides significant cache miss rate reductions for several workloads (SCP, PVC, PVR, SS, and DMR). For the two workloads (BP and BFS) where WIP increases the miss rate (5% for BP, and 2.5% for BFS), the bypassing mechanism in MeDiC is able to contain the negative effects of WIP by dynamically tuning how aggressively bypassing is performed based on the change in cache miss rate (see Section 4.3.2). We conclude that MeDiC does not hurt the overall L2 cache miss rate.

Impact of MeDiC on Queuing Latency. Figure 4.14 shows the average L2 cache queuing latency for WByp and MeDiC, compared to Baseline queuing latency. For most workloads, WByp reduces the queuing latency significantly (up to 8.7x in the case of PVR). This reduction secs/medic/results in significant performance gains for both WByp and MeDiC.

There are two applications where the queuing latency increases significantly: BFS and SSSP. We observe that when cache bypassing is applied, the GPU cores retire instructions at a much faster rate (2.33x for BFS, and 2.17x for SSSP). This increases the pressure at each shared resource, including a sharp increase in the rate of cache requests arriving at the L2 cache. This additional

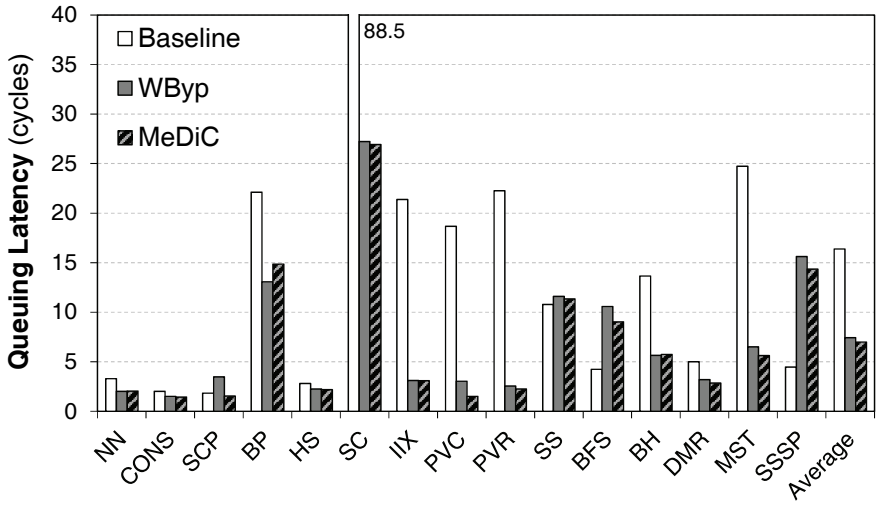


Figure 4.14. L2 queuing latency for warp-type-aware bypassing and MeDiC, compared to Baseline L2 queuing latency.

backpressure secs/medic/results in higher L2 cache queuing latencies for both applications.

When all three mechanisms in MeDiC (bypassing, cache insertion, and memory scheduling) are combined, we observe that the queuing latency reduces even further. This additional reduction occurs because the cache insertion mechanism in MeDiC reduces the cache miss rate. We conclude that in general, MeDiC significantly alleviates the L2 queuing bottleneck.

Impact of MeDiC on Row Buffer Locality. Another possible downside of cache bypassing is that it may increase the number of requests serviced by DRAM, which in turn can affect DRAM row buffer locality. Figure 4.15 shows the row buffer hit rate for WMS and MeDiC, compared to the Baseline hit rate.

Compared to Baseline, WMS has a negative effect on the row buffer locality of six applications (NN, BP, PVR, SS, BFS, and SSSP), and a positive effect on seven applications (CONS, SCP, HS, PVC, BH, DMR, and MST). We observe that even though the row buffer locality of some applications decreases, the overall performance improves, as the memory scheduler prioritizes requests from warps that are more sensitive to long memory latencies. Additionally, prioritizing requests from warps that send a small number of memory requests (mostly-hit warps) over warps that send a large number of memory requests (mostly-miss warps) allows more time for mostly-miss warps to batch requests together, improving their row buffer locality. Prior work on GPU

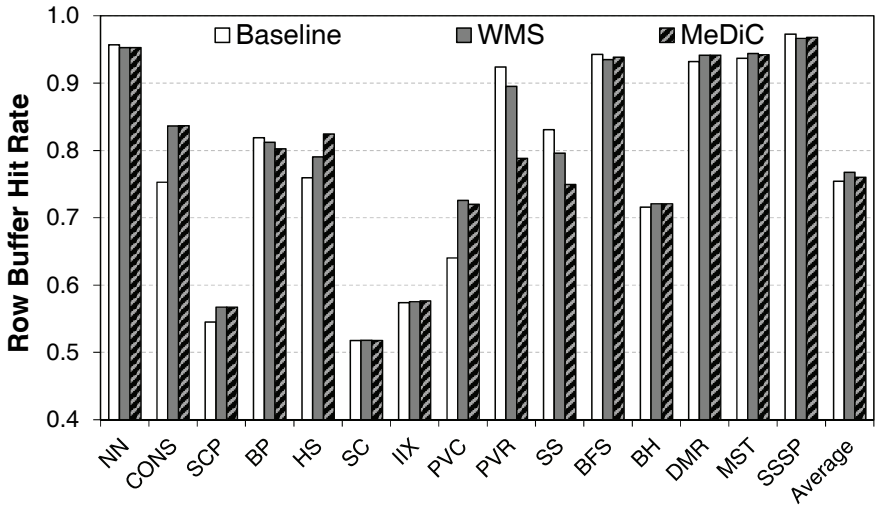


Figure 4.15. Row buffer hit rate of warp-type-aware memory scheduling and MeDiC, compared to Baseline.

memory scheduling [33] has observed similar behavior, where batching requests together allows GPU requests to benefit more from row buffer locality.

4.5.4. Identifying Reuse in GPGPU Applications

While WByp bypasses warps that have low cache utility, it is possible that some cache blocks fetched by these bypassed warps get accessed frequently. Such a frequently-accessed cache block may be needed later by a mostly-hit warp, and thus leads to an extra cache miss (as the block bypasses the cache). To remedy this, we add a mechanism to MeDiC that ensures all high-reuse cache blocks still get to access the cache. The key idea, building upon the state-of-the-art mechanism for block-level reuse [334], is to use a Bloom filter to track the high-reuse cache blocks, and to use this filter to override bypassing decisions. We call this combined design **MeDiC-reuse**.

Figure 4.16 shows that MeDiC-reuse suffers 16.1% performance degradation over MeDiC. There are two reasons behind this degradation. First, we observe that MeDiC likely implicitly captures blocks with high reuse, as these blocks tend to belong to all-hit and mostly-hit warps. Second, we observe that several GPGPU applications contain access patterns that cause severe false positive aliasing within the Bloom filter used to implement EAF and MeDiC-reuse. This leads to some low reuse cache accesses from mostly-miss and all-miss warps taking up cache space

unnecessarily, resulting in cache thrashing. We conclude that MeDiC likely implicitly captures the high reuse cache blocks that are relevant to improving memory divergence (and thus performance). However, there may still be room for other mechanisms that make the best of block-level cache reuse and warp-level heterogeneity in making caching decisions.

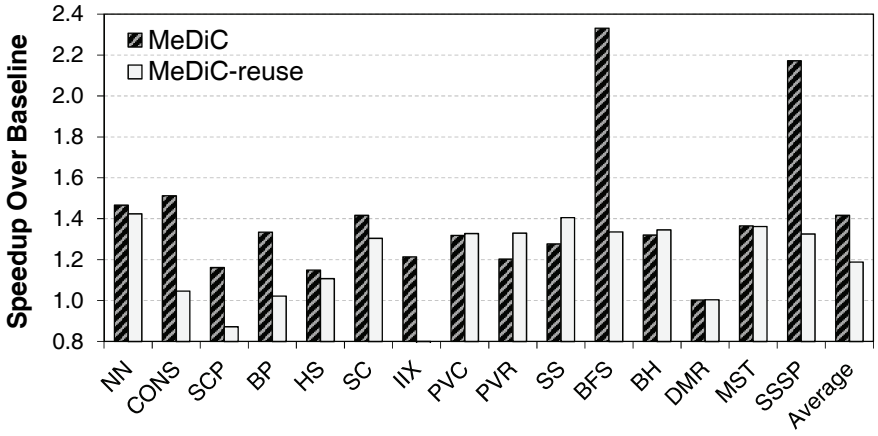


Figure 4.16. Performance of MeDiC with Bloom filter based reuse detection mechanism from the EAF cache [334].

4.5.5. Hardware Cost

MeDiC requires additional metadata storage in two locations. First, each warp needs to maintain its own hit ratio. This can be done by adding 22 bits to the metadata of each warp: two 10-bit counters to track the number of L2 cache hits and the number of L2 cache accesses, and 2 bits to store the warp type.[11] To efficiently account for overflow, the two counters that track L2 hits and L2 accesses are shifted right when the most significant bit of the latter counter is set. Additionally, the metadata for each cache line contains two bits, in order to annotate the warp type for the cache insertion policy. The total storage needed in the cache is $2 \times$ *NumCacheLines* bits. In all, MeDiC comes at a cost of 5.1 kB, or less than 1% of the L2 cache size.

To evaluate the trade-off of storage overhead, we evaluate a GPU where this overhead is converted into additional L2 cache space for the baseline GPU. We conservatively increase the L2 capacity by 5%, and find that this additional cache capacity does not improve the performance of

[11] We combine the mostly-miss and all-miss categories into a single warp type value, because we perform the same actions on both types of warps.

any of our workloads by more than 1%. As we discuss in the chapter, contention due to warp interference and divergence, and not due to cache capacity, is the root cause behind the performance bottlenecks that MeDiC alleviates. We conclude that MeDiC can deliver significant performance improvements with very low overhead.

4.6. MeDiC: Conclusion

Warps from GPGPU applications exhibit heterogeneity in their memory divergence behavior at the shared L2 cache within the GPU. We find that (1) some warps benefit significantly from the cache, while others make poor use of it; (2) such divergence behavior for a warp tends to remain stable for long periods of the warp's execution; and (3) the impact of memory divergence can be amplified by the high queuing latencies at the L2 cache.

We propose *Memory Divergence Correction* (MeDiC), whose key idea is to identify memory divergence heterogeneity in hardware and use this information to drive cache management and memory scheduling, by prioritizing warps that take the greatest advantage of the shared cache. To achieve this, MeDiC consists of three *warp-type-aware* components for (1) cache bypassing, (2) cache insertion, and (3) memory scheduling. MeDiC delivers significant performance and energy improvements over multiple previously proposed policies, and over a state-of-the-art GPU cache management technique. We conclude that exploiting inter-warp heterogeneity is effective, and hope future works explore other ways of improving systems based on this key observation.

Chapter 5

Reducing Inter-application Interference with Staged Memory Scheduling

As the number of cores continues to increase in modern chip multiprocessor (CMP) systems, the DRAM memory system is becoming a critical shared resource. Memory requests from multiple cores interfere with each other, and this inter-application interference is a significant impediment to individual application and overall system performance. Previous work on application-aware memory scheduling [197, 198, 260, 261] has addressed the problem by making the memory controller aware of application characteristics and appropriately prioritizing memory requests to improve system performance and fairness.

Recent systems [55, 161, 268] present an additional challenge by introducing integrated graphics processing units (GPUs) on the same die with CPU cores. GPU applications typically demand significantly more memory bandwidth than CPU applications due to the GPU's capability of executing a large number of parallel threads. GPUs use single-instruction multiple-data (SIMD) pipelines to concurrently execute multiple threads, where a batch of threads running the same instruction is called a wavefront or warp. When a wavefront stalls on a memory instruction, the GPU core hides this memory access latency by switching to another wavefront to avoid stalling the pipeline. Therefore, there can be thousands of outstanding memory requests from across all of

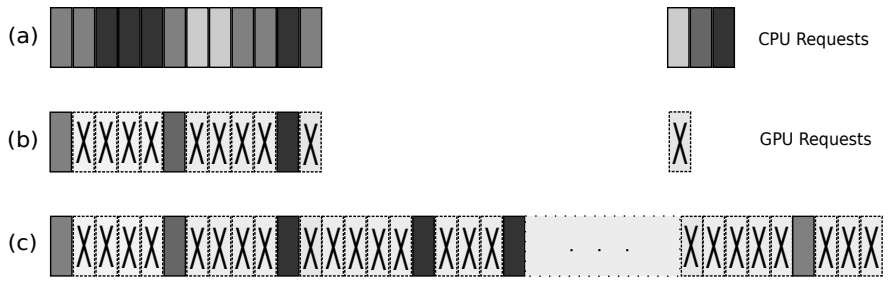


Figure 5.1. Limited visibility example. (a) CPU-only information, (b) Memory controller's visibility, (c) Improved visibility

the wavefronts. This is fundamentally more memory intensive than CPU memory traffic, where each CPU application has a much smaller number of outstanding requests due to the sequential execution model of CPUs.

Recent memory scheduling research has focused on memory interference between applications in CPU-only scenarios. These past proposals are built around a single centralized request buffer at each memory controller (MC). The scheduling algorithm implemented in the memory controller analyzes the stream of requests in the centralized request buffer to determine application memory characteristics, decides on a priority for each core, and then enforces these priorities. Observable memory characteristics may include the number of requests that result in row-buffer hits, the bank-level parallelism of each core, memory request rates, overall fairness metrics, and other information. Figure 5.1(a) shows the CPU-only scenario where the request buffer only holds requests from the CPUs. In this case, the memory controller sees a number of requests from the CPUs and has visibility into their memory behavior. On the other hand, when the request buffer is shared between the CPUs and the GPU, as shown in Figure 5.1(b), the large volume of requests from the GPU occupies a significant fraction of the memory controller's request buffer, thereby limiting the memory controller's visibility of the CPU applications' memory behaviors.

One approach to increasing the memory controller's visibility across a larger window of memory requests is to increase the size of its request buffer. This allows the memory controller to observe more requests from the CPUs to better characterize their memory behavior, as shown in Figure 5.1(c). For instance, with a large request buffer, the memory controller can identify and ser-

vice multiple requests from one CPU core to the same row such that they become row-buffer hits, however, with a small request buffer as shown in Figure 5.1(b), the memory controller may not even see these requests at the same time because the GPU's requests have occupied the majority of the entries.

Unfortunately, very large request buffers impose significant implementation challenges including the die area for the larger structures and the additional circuit complexity for analyzing so many requests, along with the logic needed for assignment and enforcement of priorities. Therefore, while building a very large, centralized memory controller request buffer could lead to good memory scheduling decisions, the approach is unattractive due to the resulting area, power, timing and complexity costs.

In this work, we propose the Staged Memory Scheduler (SMS), a decentralized architecture for memory scheduling in the context of integrated multi-core CPU-GPU systems. The key idea in SMS is to decouple the various functional requirements of memory controllers and partition these tasks across several simpler hardware structures which operate in a staged fashion. The three primary functions of the memory controller, which map to the three stages of our proposed memory controller architecture, are:

1. Detection of basic within-application memory characteristics (e.g., row-buffer locality).

2. Prioritization across applications (CPUs and GPU) and enforcement of policies to reflect the priorities.

3. Low-level command scheduling (e.g., activate, precharge, read/write), enforcement of device timing constraints (e.g., t_{RAS}, t_{FAW}, etc.), and resolving resource conflicts (e.g., data bus arbitration).

Our specific SMS implementation makes widespread use of distributed FIFO structures to maintain a very simple implementation, but at the same time SMS can provide fast service to low memory-intensity (likely latency sensitive) applications and effectively exploit row-buffer locality and bank-level parallelism for high memory-intensity (bandwidth demanding) applications. While

SMS provides a specific implementation, our staged approach for memory controller organization provides a general framework for exploring scalable memory scheduling algorithms capable of handling the diverse memory needs of integrated CPU-GPU systems of the future.

This work makes the following contributions:

- We identify and present the challenges posed to existing memory scheduling algorithms due to the highly memory-bandwidth-intensive characteristics of GPU applications.
- We propose a new decentralized, *multi-stage* approach to memory scheduling that effectively handles the interference caused by bandwidth-intensive applications, while simplifying the hardware implementation.
- We evaluate our approach against four previous memory scheduling algorithms [197, 198, 261, 317] across a wide variety workloads and CPU-GPU systems and show that it provides better performance and fairness. As an example, our evaluations on a CPU-GPU system show that SMS improves system performance by 41.2% and fairness by 4.8× across 105 multi-programmed workloads on a 16-CPU/1-GPU, four memory controller system, compared to the best previous memory scheduler TCM [198].

5.1. Background

In this section, we re-iterate DRAM organization and discuss how past research attempted to deal with the challenges of providing performance and fairness for modern memory systems.

5.1.1. Main Memory Organization

DRAM is organized as two-dimensional arrays of bitcells. Reading or writing data to DRAM requires that a row of bitcells from the array first be read into a row buffer. This is required because the act of reading the row destroys the row's contents, and so a copy of the bit values must be kept (in the row buffer). Reads and writes operate directly on the row buffer. Eventually the row is "closed" whereby the data in the row buffer are written back into the DRAM array. Accessing data already loaded in the row buffer, also called a row buffer hit, incurs a shorter latency than when the corresponding row must first be "opened" from the DRAM array. A modern memory controller (MC) must orchestrate the sequence of commands to open, read, write and close rows. Servicing requests in an order that increases row-buffer hits tends to improve overall throughput by reducing the average latency to service requests. The MC is also responsible for enforcing a wide variety of timing constraints imposed by modern DRAM standards (e.g., DDR3) such as limiting the rate of page-open operations (t_{FAW}) and ensuring a minimum amount of time between writes and reads (t_{WTR}).

Each two dimensional array of DRAM cells constitutes a bank, and a group of banks form a rank. All banks within a rank share a common set of command and data buses, and the memory controller is responsible for scheduling commands such that each bus is used by only one bank at a time. Operations on multiple banks may occur in parallel (e.g., opening a row in one bank while reading data from another bank's row buffer) so long as the buses are properly scheduled and any other DRAM timing constraints are honored. A memory controller can improve memory system throughput by scheduling requests such that bank-level parallelism or BLP (i.e., the number of banks simultaneously busy responding to commands) is increased. A memory system implementation may support multiple independent memory channels (each with its own ranks and banks) to further increase the number of memory requests that can be serviced at the same time. A key challenge in the implementation of modern, high-performance memory controllers is to effectively improve system performance by maximizing both row-buffer hits and BLP while simultaneously providing fairness among multiple CPUs and the GPU.

5.1.2. Memory Scheduling

Accessing off-chip memory is one of the major bottlenecks in microprocessors. Requests that miss in the last level cache incur long latencies, and as multi-core processors increase the number of CPUs, the problem gets worse because all of the cores must share the limited off-chip memory bandwidth. The large number of requests greatly increases contention for the memory data and command buses. Since a bank can only process one command at a time, the large number of requests also increases bank contention where requests must wait for busy banks to finish servicing other requests. A request from one core can also cause a row buffer containing data for another core to be closed, thereby reducing the row-buffer hit rate of that other core (and vice-versa). All of these effects increase the latency of memory requests by both increasing queuing delays (time spent waiting for the memory controller to start servicing a request) and DRAM device access delays (due to decreased row-buffer hit rates and bus contention).

The memory controller is responsible for buffering and servicing memory requests from the different cores and the GPU. Typical implementations make use of a memory request buffer to hold and keep track of all in-flight requests. Scheduling logic then decides which requests should be serviced, and issues the corresponding commands to the DRAM devices. Different memory scheduling algorithms may attempt to service memory requests in an order different than the order in which the requests arrived at the memory controller, in order to increase row-buffer hit rates, bank level parallelism, fairness, or achieve other goals.

5.1.3. Memory Scheduling in CPU-only Systems

Memory scheduling algorithms improve system performance by reordering memory requests to deal with the different constraints and behaviors of DRAM. The first-ready-first-come-first-serve (FR-FCFS) [317] algorithm attempts to schedule requests that result in row-buffer hits (first-ready), and otherwise prioritizes older requests (FCFS). FR-FCFS increases DRAM throughput, but it can cause fairness problems by under-servicing applications with low row-buffer locality. Several application-aware memory scheduling algorithms [197, 198, 260, 261] have been proposed to bal-

ance both performance and fairness. Parallelism-aware Batch Scheduling (PAR-BS) [261] batches requests based on their arrival times (older requests batched first). Within a batch, applications are ranked to preserve bank-level parallelism (BLP) within an application's requests. More recently, ATLAS [197] proposes prioritizing applications that have received the least memory service. As a result, applications with low memory intensities, which typically attain low memory service, are prioritized. However, applications with high memory intensities are deprioritized and hence slowed down significantly, resulting in unfairness. The most recent work on application-aware memory scheduling, Thread Cluster Memory scheduling (TCM) [198], addresses this unfairness problem. TCM first clusters applications into low and high memory-intensity clusters based on their memory intensities. TCM always prioritizes applications in the low memory-intensity cluster, however, among the high memory-intensity applications it shuffles request priorities to prevent unfairness.

5.1.4. Characteristics of Memory Accesses from GPUs

A typical CPU application only has a relatively small number of outstanding memory requests at any time. The size of a processor's instruction window bounds the number of misses that can be simultaneously exposed to the memory system. Branch prediction accuracy limits how large the instruction window can be usefully increased. In contrast, GPU applications have very different access characteristics, generating many more memory requests than CPU applications. A GPU application can consist of many thousands of parallel threads, where memory stalls on one group of threads can be hidden by switching execution to one of the many other groups of threads.

Figure 5.2 (a) shows the memory request rates for a representative subset of our GPU applications and the most memory-intensive SPEC2006 (CPU) applications, as measured by memory requests per thousand cycles (see Section 5.3.5 for simulation methodology descriptions) when each application runs alone on the system. The raw bandwidth demands of the GPU applications are often multiple times higher than the SPEC benchmarks. Figure 5.2 (b) shows the row-buffer hit rates (also called row-buffer locality or RBL). The GPU applications show consistently high

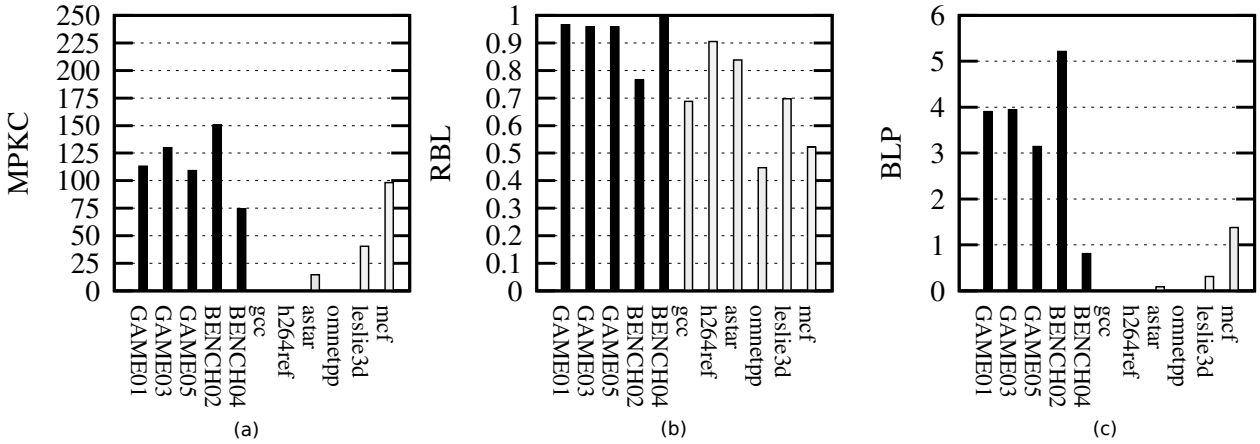


Figure 5.2. GPU memory characteristic. (a) Memory-intensity, measured by memory requests per thousand cycles, (b) Row buffer locality, measured by the fraction of accesses that hit in the row buffer, and (c) Bank-level parallelism.

levels of RBL, whereas the SPEC benchmarks exhibit more variability. The GPU programs have high levels of spatial locality, often due to access patterns related to large sequential memory accesses (e.g., frame buffer updates). Figure 5.2(c) shows the BLP for each application, with the GPU programs consistently making use of far banks at the same time.

In addition to the high-intensity memory traffic of GPU applications, there are other properties that distinguish GPU applications from CPU applications. The TCM [198] study observed that CPU applications with streaming access patterns typically exhibit high RBL but low BLP, while applications with less uniform access patterns typically have low RBL but high BLP. In contrast, GPU applications have *both* high RBL and high BLP. The combination of high memory intensity, high RBL and high BLP means that the GPU will cause significant interference to other applications across all banks, especially when using a memory scheduling algorithm that preferentially favors requests that result in row-buffer hits.

5.1.5. What Has Been Done in the GPU?

As opposed to CPU applications, GPU applications are not very latency sensitive as there are a large number of independent threads to cover long memory latencies. However, the GPU requires a significant amount of bandwidth far exceeding even the most memory-intensive CPU applications.

As a result, a GPU memory scheduler [226] typically needs a large request buffer that is capable of request coalescing (i.e., combining multiple requests for the same block of memory into a single combined request [274]). Furthermore, since GPU applications are bandwidth intensive, often with streaming access patterns, a policy that maximizes the number of row-buffer hits is effective for GPUs to maximize overall throughput. As a result, FR-FCFS with a large request buffer tends to perform well for GPUs [41]. In view of this, previous work [397] designed mechanisms to reduce the complexity of row-hit first based (FR-FCFS) scheduling.

5.2. Challenges with Existing Memory Controllers

5.2.1. The Need for Request Buffer Capacity

The results from Figure 5.2 showed that GPU applications have very high memory intensities. As discussed in Section 2.2.1, the large number of GPU memory requests occupy many of the memory controller's request buffer entries, thereby making it very difficult for the memory controller to properly determine the memory access characteristics of each of the CPU applications. Figure 5.3 shows the performance impact of increasing the memory controller's request buffer size for a variety of memory scheduling algorithms (full methodology details can be found in Section 5.3.5) for a 16-CPU/1-GPU system. By increasing the size of the request buffer from 64 entries to 256 entries,[1] previously proposed memory controller algorithms can gain up to 63.6% better performance due to this improved visibility.

5.2.2. Implementation Challenges in Providing Request Buffer Capacity

The results above show that when the memory controller has enough visibility across the global memory request stream to properly characterize the behaviors of each core, a sophisticated algorithm like TCM can be effective at making good scheduling decisions. Unfortunately, implementing a sophisticated algorithm like TCM over such a large scheduler introduces very significant implementation challenges. For all algorithms that use a centralized request buffer and priori-

[1] For all sizes, half of the entries are reserved for the CPU requests.

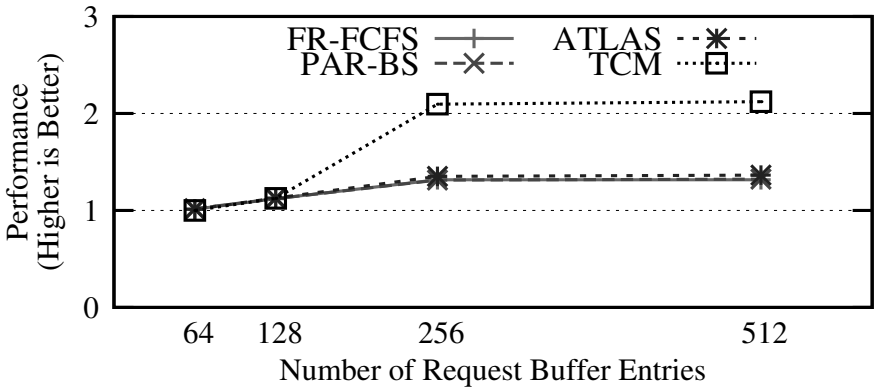


Figure 5.3. Performance at different request buffer sizes

tize requests that result in row-buffer hits (FR-FCFS, PAR-BS, ATLAS, TCM), associative logic (CAMs) will be needed for each entry to compare its requested row against currently open rows in the DRAM banks. For all algorithms that prioritize requests based on rank/age (FR-FCFS, PAR-BS, ATLAS, TCM), a large comparison tree is needed to select the highest ranked/oldest request from all request buffer entries. The size of this comparison tree grows with request buffer size. Furthermore, in addition to this logic for reordering requests and enforcing ranking/age, TCM also requires additional logic to continually monitor each CPU's last-level cache MPKI rate (note that a CPU's instruction count is not typically available at the memory controller), each core's RBL which requires additional *shadow row buffer index* tracking [90, 92], and each core's BLP.

Apart from the logic required to implement the policies of the specific memory scheduling algorithms, all of these memory controller designs need additional logic to enforce DDR timing constraints. Note that different timing constraints will apply depending on the *state* of each memory request. For example, if a memory request's target bank currently has a different row loaded in its row buffer, then the memory controller must ensure that a precharge (row close) command is allowed to issue to that bank (e.g., has t_{RAS} elapsed since the row was opened?), but if the row is already closed, then different timing constraints will apply. For each request buffer entry, the memory controller will determine whether or not the request can issue a command to the DRAM based on the current state of the request and the current state of the DRAM system. That is, *every* request buffer entry (i.e., all 256) needs an independent instantiation of the DDR compliance-

checking logic (including data and command bus availability tracking). This type of monolithic memory controller effectively implements a large out-of-order scheduler; note that typical instruction schedulers in modern out-of-order processors only have about 32-64 entries [107]. Even after accounting for the clock speed differences between CPU core and DRAM command frequencies, it is very difficult to implement a fully-associative[2], age-ordered/prioritized, out-of-order scheduler with 256-512 entries [285].

5.3. The Staged Memory Scheduler

The proposed Staged Memory Scheduler (SMS) is structured to reflect the primary functional tasks of the memory scheduler. Below, we first describe the overall SMS algorithm, explain additional implementation details, step through the rationale for the design, and then walk through the hardware implementation.

5.3.1. The SMS Algorithm

Batch Formation. The first stage of SMS consists of several simple FIFO structures, one per *source* (i.e., a CPU core or the GPU). Each request from a given source is initially inserted into its respective FIFO upon arrival at the memory controller. A *batch* is simply one or more memory requests from the same source that access the same DRAM row. That is, all requests within a batch, except perhaps for the first one, would be row-buffer hits if scheduled consecutively. A batch is complete or *ready* when an incoming request accesses a different row, when the oldest request in the batch has exceeded a threshold age, or when the FIFO is full. Ready batches may then be considered by the second stage of the SMS.

Batch Scheduler. The batch formation stage has combined individual memory requests into batches of row-buffer hitting requests. The next stage, the batch scheduler, deals directly with batches, and therefore need not worry about scheduling to optimize for row-buffer locality. Instead, the batch scheduler can focus on higher-level policies regarding inter-application interfer-

[2]Fully associative in the sense that a request in *any* one of the request buffer entries could be eligible to issue in a given cycle.

ence and fairness. The goal of the batch scheduler is to prioritize batches from applications that are latency critical, while making sure that bandwidth-intensive applications (e.g., the GPU) still make reasonable progress.

The batch scheduler operates in two states: pick and drain. In the pick state, the batch scheduler considers each FIFO from the batch formation stage. For each FIFO that contains a *ready* batch, the batch scheduler picks one batch based on a balance of shortest-job first (SJF) and round-robin principles. For SJF, the batch scheduler chooses the core (or GPU) with the fewest total memory requests across all three stages of the SMS. SJF prioritization reduces average request service latency, and it tends to favor latency-sensitive applications, which tend to have fewer total requests. The other component of the batch scheduler is a round-robin policy that simply cycles through each of the per-source FIFOs ensuring that high memory-intensity applications receive adequate service. Overall, the batch scheduler chooses the SJF policy with a probability of p, and the round-robin policy otherwise.

After picking a batch, the batch scheduler enters a drain state where it forwards the requests from the selected batch to the final stage of the SMS. The batch scheduler simply dequeues one request per cycle until all requests from the batch have been removed from the selected batch formation FIFO. At this point, the batch scheduler re-enters the pick state to select the next batch.

DRAM Command Scheduler. The last stage of the SMS is the DRAM command scheduler (DCS). The DCS consists of one FIFO queue per DRAM bank (e.g., eight banks/FIFOs for DDR3). The drain phase of the batch scheduler places the memory requests directly into these FIFOs. Note that because batches are moved into the DCS FIFOs one batch at a time, any row-buffer locality within a batch is preserved within a DCS FIFO. At this point, any higher-level policy decisions have already been made by the batch scheduler, therefore, the DCS can simply focus on issuing low-level DRAM commands and ensuring DDR protocol compliance.

On any given cycle, the DCS only considers the requests at the *head* of each of the per-bank FIFOs. For each request, the DCS determines whether that request can issue a command based on the request's current row-buffer state (i.e., is the row buffer already open with the requested

row, closed, or open with the wrong row?) and the current DRAM state (e.g., time elapsed since a row was opened in a bank, data bus availability). If more than one request is eligible to issue a command, the DCS simply arbitrates in a round-robin fashion.

5.3.2. Additional Algorithm Details

Batch Formation Thresholds. The batch formation stage holds requests in the per-source FIFOs until a complete batch is ready. This could unnecessarily delay requests as the batch will not be marked ready until a request to a *different* row arrives at the memory controller, or the FIFO size has been reached. This additional queuing delay can be particularly devastating for low-intensity, latency-sensitive applications.

SMS considers an application's memory intensity in forming batches. For applications with low memory-intensity ($<$1 MPKC), SMS completely bypasses the batch formation and batch scheduler, and forwards requests directly to the DCS per-bank FIFOs. For these highly sensitive applications, such a bypass policy minimizes the delay to service their requests. Note that this bypass operation will not interrupt an on-going drain from the batch scheduler, which ensures that any separately scheduled batches maintain their row-buffer locality.

For medium memory-intensity (1-10 MPKC) and high memory-intensity ($>$10 MPKC) applications, the batch formation stage uses age thresholds of 50 and 200 cycles, respectively. That is, regardless of how many requests are in the current batch, when the oldest request's age exceeds the threshold, the entire batch is marked ready (and consequently, any new requests that arrive, even if accessing the same row, will be grouped into a new batch). Note that while TCM uses the MPKI metric to classify memory intensity, SMS uses misses per thousand *cycles* (MPKC) since the per-application instruction counts are not typically available in the memory controller. While it would not be overly difficult to expose this information, this is just one less implementation overhead that SMS can avoid.

Global Bypass. As described above, low memory-intensity applications can bypass the entire batch formation and scheduling process and proceed directly to the DCS. Even for high memory-

intensity applications, if the memory system is lightly loaded (e.g., if this is the only application running on the system right now), then the SMS will allow all requests to proceed directly to the DCS. This bypass is enabled whenever the total number of in-flight requests (across *all* sources) in the memory controller is less than sixteen requests.

Round-Robin Probability. As described above, the batch scheduler uses a probability of p to schedule batches with the SJF policy and the round-robin policy otherwise. Scheduling batches in a round-robin order can ensure fair progress from high-memory intensity applications. Our experimental results show that setting p to 90% (10% using the round-robin policy) provides a good performance-fairness trade-off for SMS.

5.3.3. SMS Rationale

In-Order Batch Formation. It is important to note that batch formation occurs in the order of request arrival. This potentially sacrifices some row-buffer locality as requests to the same row may be interleaved with requests to other rows. We considered many variations of batch formation that allowed out-of-order grouping of requests to maximize the length of a run of row-buffer hitting requests, but the overall performance benefit was not significant. First, constructing very large batches of row-buffer hitting requests can introduce significant unfairness as other requests may need to wait a long time for a bank to complete its processing of a long run of row-buffer hitting requests [187]. Second, row-buffer locality across batches may still be exploited by the DCS. For example, consider a core that has three batches accessing row X, row Y, and then row X again. If X and Y map to different DRAM banks, say banks A and B, then the batch scheduler will send the first and third batches (row X) to bank A, and the second batch (row Y) to bank B. Within the DCS's FIFO for bank A, the requests for the first and third batches will all be one after the other, thereby exposing the row-buffer locality across batches despite the requests appearing "out-of-order" in the original batch formation FIFOs.

In-Order Batch Scheduling. Due to contention and back-pressure in the system, it is possible that a FIFO in the batch formation stage contains more than one valid batch. In such a case, it

could be desirable for the batch scheduler to pick one of the batches not currently at the head of the FIFO. For example, the bank corresponding to the head batch may be busy while the bank for another batch is idle. Scheduling batches out of order could decrease the service latency for the later batches, but in practice it does not make a big difference and adds significant implementation complexity. It is important to note that even though batches are dequeued from the batch formation stage in arrival order per FIFO, the request order *between* the FIFOs may still slip relative to each other. For example, the batch scheduler may choose a recently arrived (and formed) batch from a high-priority (i.e., latency-sensitive) source even though an older, larger batch from a different source is ready.

In-Order DRAM Command Scheduling. For each of the per-bank FIFOs in the DCS, the requests are already grouped by row-buffer locality (because the batch scheduler drains an entire batch at a time), and globally ordered to reflect per-source priorities. Further reordering at the DCS would likely just undo the prioritization decisions made by the batch scheduler. Like the batch scheduler, the in-order nature of each of the DCS per-bank FIFOs does not prevent out-of-order scheduling at the global level. A CPU's requests may be scheduled to the DCS in arrival order, but the requests may get scattered across different banks, and the issue order among banks may slip relative to each other.

5.3.4. Hardware Implementation

The staged architecture of SMS lends directly to a low-complexity hardware implementation. Figure 5.4 illustrates the overall hardware organization of SMS.

Batch Formation. The batch formation stage consists of little more than one FIFO per source (CPU or GPU). Each FIFO maintains an extra register that records the row index of the last request, so that any incoming request's row index can be compared to determine if the request can be added to the existing batch. Note that this requires only a single comparator (used only once at insertion) per FIFO. Contrast this to a conventional monolithic request buffer where comparisons on *every* request buffer entry (which is much larger than the number of FIFOs that SMS uses) must

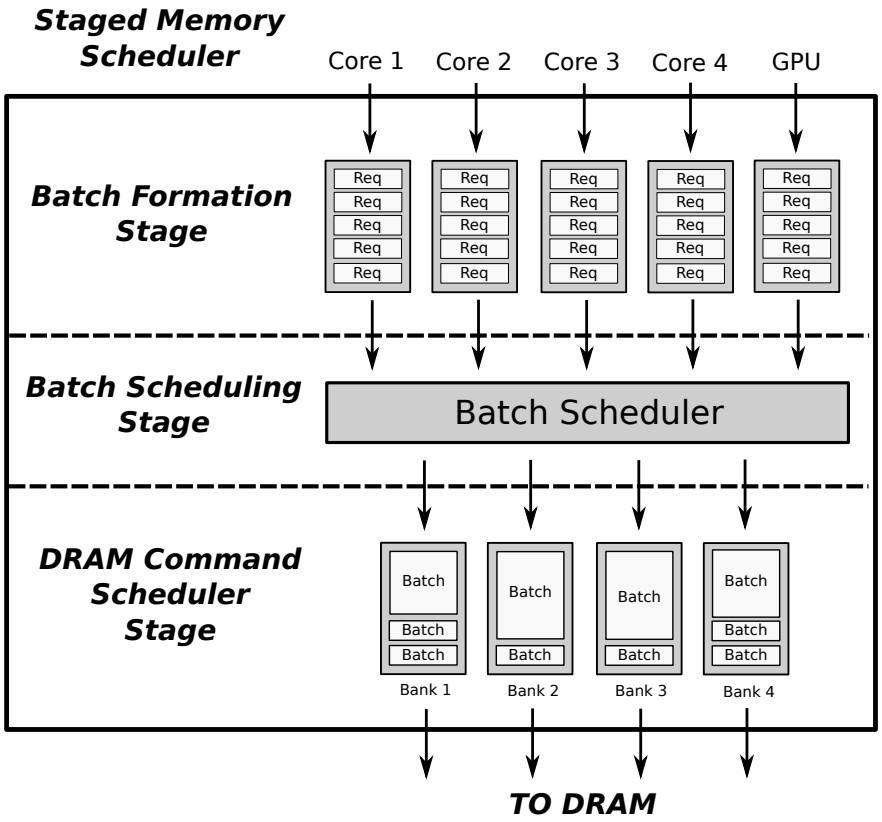


Figure 5.4. The design of SMS

be made, potentially against all currently open rows across all banks.

Batch Scheduler. The batch scheduling stage consists primarily of combinatorial logic to implement the batch picking rules. When using the SJF policy, the batch scheduler only needs to pick the batch corresponding to the source with the fewest in-flight requests, which can be easily performed with a tree of MIN operators. Note that this tree is relatively shallow since it only grows as a function of the number of FIFOs. Contrast this to the monolithic scheduler where the various ranking trees grow as a function of the total number of entries.

DRAM Command Scheduler. The DCS stage consists of the per-bank FIFOs. The logic to track and enforce the various DDR timing and power constraints is identical to the case of the monolithic scheduler, but the scale is drastically different. The DCS's DDR command-processing logic only considers the requests at the head of each of the per-bank FIFOs (eight total for DDR3),

whereas the monolithic scheduler requires logic to consider *every* request buffer entry (hundreds).

Overall Configuration and Hardware Cost. The final configuration of SMS that we use in this dissertation consists of the following hardware structures. The batch formation stage uses ten-entry FIFOs for each of the CPU cores, and a twenty-entry FIFO for the GPU. The DCS uses a fifteen-entry FIFO for each of the eight DDR3 banks. For sixteen cores and a GPU, the aggregate capacity of all of these FIFOs is 300 requests, although at any point in time, the SMS logic can only consider or act on a small subset of the entries (i.e., the seventeen at the heads of the batch formation FIFOs and the eight at the heads of the DCS FIFOs). In addition to these primary structures, there are a small handful of bookkeeping counters. One counter per source is needed to track the number of in-flight requests; each counter is trivially managed as it only needs to be incremented when a request arrives at the memory controller, and then decremented when the request is complete. Counters are also needed to track per-source MPKC rates for memory-intensity classification, which are incremented when a request arrives, and then periodically reset. Table 5.1 summarizes the amount of hardware overhead required for each stage of SMS.

Storage	Description	Size
Storage Overhead of Stage 1: Batch formation stage		
CPU FIFO queues	A CPU core's FIFO queue	$N_{core} \times Queue\_Size_{core} = 160$ entries
GPU FIFO queues	A GPU's FIFO queue	$N_{GPU} \times Queue\_Size_{GPU} = 20$ entries
MPKC counters	Counts per-core MPKC	$N_{core} \times log_2 MPKC_{max} = 160$ bits
Last request's row index	Stores the row index of the last request to the FIFO	$(N_{core} + N_{GPU}) \times log_2 Row\_Index\_Size = 204$ bits
Storage Overhead of Stage 2: Batch Scheduler		
CPU memory request counters	Counts the number of outstanding memory requests of a CPU core	$N_{core} \times log_2 Count_{max\_CPU} = 80$ bits
GPU memory request counter	Counts the number of outstanding memory requests of the GPU	$N_{GPU} \times log_2 Count_{max\_GPU} = 10$ bits
Storage Overhead of Stage 3: DRAM Command Scheduler		
Per-Bank FIFO queues	Contains a FIFO queue per bank	$N_{banks} \times Queue\_Size_{bank} = 120$ entries

Table 5.1. Hardware storage required for SMS

5.3.5. Experimental Methodology

We use an in-house cycle-accurate simulator to perform our evaluations. For our performance evaluations, we model a system with sixteen x86 CPU cores and a GPU. For the CPUs, we model three-wide out-of-order processors with a cache hierarchy including per-core L1 caches and a

shared, distributed L2 cache. The GPU does not share the CPU caches. Table 5.2 shows the detailed system parameters for the CPU and GPU cores. The parameters for the main memory system are listed in Table 5.2. Unless stated otherwise, we use four memory controllers (one channel per memory controller) for all experiments. In order to prevent the GPU from taking the majority of request buffer entries, we reserve half of the request buffer entries for the CPUs. To model the memory bandwidth of the GPU accurately, we perform coalescing on GPU memory requests before they are sent to the memory controller [226].

Parameter	**Setting**
CPU Clock Speed	3.2GHz
CPU ROB	128 entries
CPU L1 cache	32KB Private, 4-way
CPU L2 cache	8MB Shared, 16-way
CPU Cache Rep. Policy	LRU
GPU SIMD Width	800
GPU Texture units	40
GPU Z units	64
GPU Color units	16
Memory Controller Entries	300
Channels/Ranks/Banks	4/1/8
DRAM Row buffer size	2KB
DRAM Bus	128 bits/channel
tRCD/tCAS/tRP	8/8/8 ns
tRAS/tRC/tRRD	20/27/4 ns
tWTR/tRTP/tWR	4/4/6 ns

Table 5.2. Simulation parameters.

Workloads. We evaluate our system with a set of 105 multiprogrammed workloads, each simulated for 500 million cycles. Each workload consists of sixteen SPEC CPU2006 benchmarks and one GPU application selected from a mix of video games and graphics performance benchmarks. For each CPU benchmark, we use PIN [234, 315] with PinPoints [288] to select the representative phase. For the GPU application, we use an industrial GPU simulator to collect memory requests with detailed timing information. These requests are collected after having first been filtered through the GPU's internal cache hierarchy, therefore we do not further model any caches

for the GPU in our final hybrid CPU-GPU simulation framework.

We classify CPU benchmarks into three categories (Low, Medium, and High) based on their memory intensities, measured as last-level cache misses per thousand instructions (MPKI). Table 5.3 shows the MPKI for each CPU benchmark. Benchmarks with less than 1 MPKI are low memory-intensive, between 1 and 25 MPKI are medium memory-intensive, and greater than 25 are high memory-intensive. Based on these three categories, we randomly choose a number of benchmarks from each category to form workloads consisting of seven intensity mixes: L (All low), ML (Low/Medium), M (All medium), HL (High/Low), HML (High/Medium/Low), HM (High/Medium) and H(All high). The GPU benchmark is randomly selected for each workload without any classification.

Name	MPKI	Name	MPKI	Name	MPKI
tonto	0.01	sjeng	1.08	omnetpp	21.85
povray	0.01	gobmk	1.19	milc	21.93
calculix	0.06	gromacs	1.67	xalancbmk	22.32
perlbench	0.11	h264ref	1.86	libquantum	26.27
namd	0.11	bzip2	6.08	leslie3d	38.13
dealII	0.14	astar	7.6	soplex	52.45
wrf	0.21	hmmer	8.65	GemsFDTD	63.61
gcc	0.33	cactusADM	14.99	lbm	69.63
		sphinx3	17.24	mcf	155.30

Table 5.3. L2 Cache Misses Per Kilo-Instruction (MPKI) of 26 SPEC 2006 benchmarks.

Performance Metrics. In an integrated CPUs and GPU system like the one we evalute, To measure system performance, we use *CPU+GPU Weighted Speedup* (Eqn. 5.1), which is a sum of the CPU weighted speedup [97, 98] and the GPU speedup multiply by the weight of the GPU. In addition, we measure *Unfairness* [83, 197, 198, 373] using maximum slowdown for all the CPU cores. We report the harmonic mean instead of arithmetic mean for *Unfairness* in our evaluations since slowdown is an inverse metric of speedup.

$$CPU + GPU Weighted Speedup = \sum_{i=1}^{NCPU} \frac{IPC_i^{shared}}{IPC_i^{alone}} + WEIGHT * \frac{GPU_{FrameRate}^{shared}}{GPU_{FrameRate}^{alone}} \quad (5.1)$$

$$Unfairness = \max_i \frac{IPC_i^{alone}}{IPC_i^{shared}} \quad (5.2)$$

5.4. Qualitative Comparison with Previous Scheduling Algorithms

In this section, we compare SMS qualitatively to previously proposed scheduling policies and analyze the basic differences between SMS and these policies. The fundamental difference between SMS and previously proposed memory scheduling policies for CPU only scenarios is that the latter are designed around a single, centralized request buffer which has poor scalability and complex scheduling logic, while SMS is built around a decentralized, scalable framework.

5.4.1. First-Ready FCFS (FR-FCFS)

FR-FCFS [317] is a commonly used scheduling policy in commodity DRAM systems. A FR-FCFS scheduler prioritizes requests that result in row-buffer hits over row-buffer misses and otherwise prioritizes older requests. Since FR-FCFS unfairly prioritizes applications with high row-buffer locality to maximize DRAM throughput, prior work [197, 198, 252, 260, 261] have observed that it has low system performance and high unfairness.

5.4.2. Parallelism-aware Batch Scheduling (PAR-BS)

PAR-BS [261] aims to improve fairness and system performance. In order to prevent unfairness, it forms batches of outstanding memory requests and prioritizes the oldest batch, to avoid request starvation. To improve system throughput, it prioritizes applications with smaller number of outstanding memory requests within a batch. However, PAR-BS has two major shortcomings. First, batching could cause older GPU requests and requests of other memory-intensive CPU applications to be prioritized over latency-sensitive CPU applications. Second, as previous work [197] has also observed, PAR-BS does not take into account an application's long term memory-intensity characteristics when it assigns application priorities within a batch. This could cause memory-intensive applications' requests to be prioritized over latency-sensitive applications'

requests within a batch.

5.4.3. Adaptive per-Thread Least-Attained-Serviced Memory Scheduling (ATLAS)

ATLAS [197] aims to improve system performance by prioritizing requests of applications with lower attained memory service. This improves the performance of low memory-intensity applications as they tend to have low attained service. However, ATLAS has the disadvantage of not preserving fairness. Previous work [197, 198] have shown that simply prioritizing low memory intensity applications leads to significant slowdown of memory-intensive applications.

5.4.4. Thread Cluster Memory Scheduling (TCM)

TCM [198] is the best state-of-the-art application-aware memory scheduler providing both system throughput and fairness. It groups applications into either latency- or bandwidth-sensitive clusters based on their memory intensities. In order to achieve high system throughput and low unfairness, TCM employs different prioritization policy for each cluster. To improve system throughput, a fraction of total memory bandwidth is dedicated to latency-sensitive cluster and applications within the cluster are then ranked based on memory intensity with least memory-intensive application receiving the highest priority. On the other hand, TCM minimizes unfairness by periodically shuffling applications within a bandwidth-sensitive cluster to avoid starvation. This approach provides both high system performance and fairness in CPU-only systems. In an integrated CPU-GPU system, GPU generates a significantly larger amount of memory requests compared to CPUs and fills up the centralized request buffer. As a result, the memory controller lacks the visibility of CPU memory requests to accurately determine each application's memory access behavior. Without the visibility, TCM makes incorrect and non-robust clustering decisions, which classify some applications with high memory intensity into the latency-sensitive cluster. These misclassified applications cause interference not only to low memory intensity applications, but also to each other. Therefore, TCM causes some degradation in both system performance and fairness in an integrated CPU-GPU system. As described in Section 5.2, increasing the request buffer size is

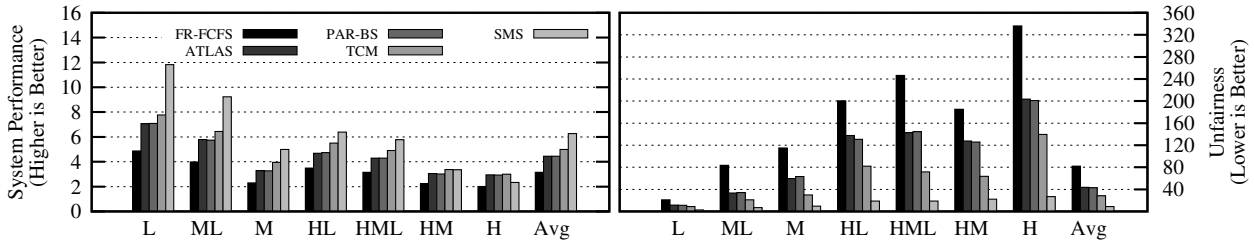


Figure 5.5. System performance, and fairness for 7 categories of workloads (total of 105 workloads)

a simple and straightforward way to gain more visibility into CPU applications' memory access behaviors. However, this approach is not scalable as we show in our evaluations (Section 4.5). In contrast, SMS provides much better system performance and fairness than TCM with the same number of request buffer entries and lower hardware cost.

5.5. Experimental Evaluation of SMS

We present the performance of five memory scheduler configurations: FR-FCFS, ATLAS, PAR-BS, TCM, and SMS on the 16-CPU/1-GPU four-memory-controller system described in Section 5.3.5. All memory schedulers use 300 request buffer entries per memory controller; this size was chosen based on the results in Figure 5.3 which showed that performance does not appreciably increase for larger request buffer sizes. Results are presented in the workload categories as described in Section 5.3.5, with workload memory intensities increasing from left to right.

Figure 5.5 shows the system performance (measured as weighted speedup) and fairness of the previously proposed algorithms and SMS, averaged across 15 workloads for each of the seven categories (105 workloads in total). Compared to TCM, which is the best previous algorithm for both system performance and fairness, SMS provides 41.2% system performance improvement and 4.8$\times$ fairness improvement. Therefore, we conclude that SMS provides better system performance and fairness than all previously proposed scheduling policies, while incurring much lower hardware cost and simpler scheduling logic.

Based on the results for each workload category, we make the following major observations: First, SMS consistently outperforms previously proposed algorithms (given the same number of

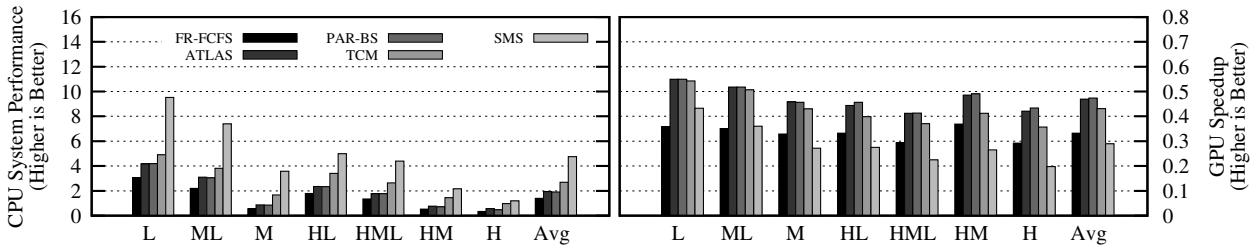


Figure 5.6. CPUs and GPU Speedup for 7 categories of workloads (total of 105 workloads)

request buffer entries), in terms of both system performance and fairness across most of the workload categories. Second, in the "H" category with only high memory-intensity workloads, SMS underperforms by 21.2%/20.7%/22.3% compared to ATLAS/PAR-BS/TCM, but SMS still provides 16.3% higher system performance compared to FR-FCFS. The main reason for this behavior is that ATLAS/PAR-BS/TCM improve performance by unfairly prioritizing certain applications over others, which is reflected by their poor fairness results. For instance, we observe that TCM misclassifies some of these high memory-intensity applications into the low memory-intensity cluster, which starves requests of applications in the high memory-intensity cluster. On the other hand, SMS preserves fairness in all workload categories by using its probabilistic round-robin policy as described in Section 5.3. As a result, SMS provides 7.6×/7.5×/5.2× better fairness relative to ATLAS/PAR-BS/TCM respectively, for the high memory-intensity category.

5.5.1. Analysis of CPU and GPU Performance

In this section, we study the performance of the CPU system and the GPU system separately. Figure 5.6 shows CPU-only weighted speedup and GPU speedup. Two major observations are in order. First, SMS gains 1.76× improvement in CPU system performance over TCM. Second, SMS achieves this 1.76× CPU performance improvement while delivering similar GPU performance as the FR-FCFS baseline.[3] The results show that TCM (and the other algorithms) end up allocating far more bandwidth to the GPU, at significant performance and fairness cost to the CPU applications. SMS appropriately deprioritizes the memory bandwidth intensive GPU application in order

[3]Note that our GPU Speedup metric is defined with respect to the performance of the GPU benchmark running on the system by itself. In all cases, the relative speedup reported is much less than 1.0 because the GPU must now share memory bandwidth with 16 CPUs.

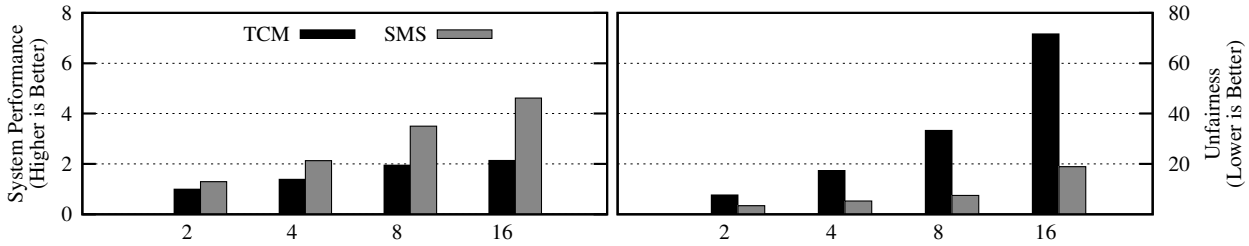


Figure 5.7. SMS vs TCM on a 16 CPU/1 GPU, 4 memory controller system with varying the number of cores

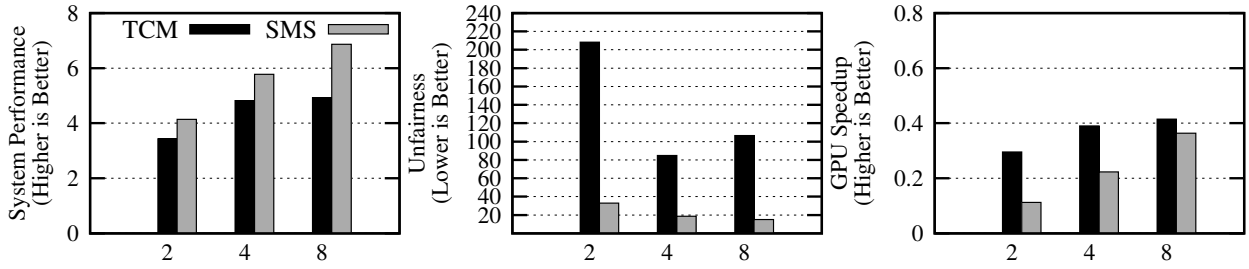


Figure 5.8. SMS vs TCM on a 16 CPU/1 GPU system with varying the number of channels

to enable higher CPU performance and overall system performance, while preserving fairness. Previously proposed scheduling algorithms, on the other hand, allow the GPU to hog memory bandwidth and significantly degrade system performance and fairness (Figure 5.5).

5.5.2. Scalability with Cores and Memory Controllers

Figure 5.7 compares the performance and fairness of SMS against TCM (averaged over 75 workloads[4]) with the same number of request buffers, as the number of cores is varied. We make the following observations: First, SMS continues to provide better system performance and fairness than TCM. Second, the system performance gains and fairness gains increase significantly as the number of cores and hence, memory pressure is increased. SMS's performance and fairness benefits are likely to become more significant as core counts in future technology nodes increase.

Figure 5.8 shows the system performance and fairness of SMS compared against TCM as the number of memory channels is varied. For this, and all subsequent results, we perform our evaluations on 60 workloads from categories that contain high memory-intensity applications. We

[4]We use 75 randomly selected workloads per core count. We could not use the same workloads/categorizations as earlier because those were for 16-core systems, whereas we are now varying the number of cores.

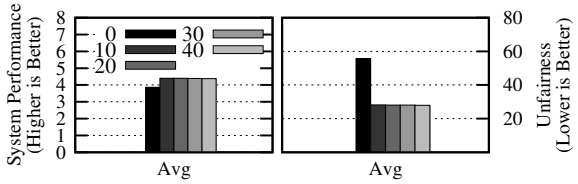


Figure 5.9. SMS sensitivity to batch Size

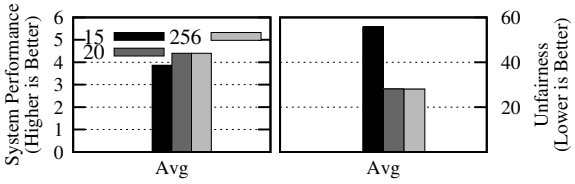


Figure 5.10. SMS sensitivity to DCS FIFO Size

observe that SMS scales better as the number of memory channels increases. As the performance gain of TCM diminishes when the number of memory channels increases from 4 to 8 channels, SMS continues to provide performance improvement for both CPU and GPU.

5.5.3. Sensitivity to SMS Design Parameters

Effect of Batch Formation. Figure 5.9 shows the system performance and fairness of SMS as the maximum batch size varies. When the batch scheduler can forward individual requests to the DCS, the system performance and fairness drops significantly by 12.3% and 1.9× compared to when it uses a maximum batch size of ten. The reasons are twofold: First, intra-application row-buffer locality is not preserved without forming requests into batches and this causes performance degradation due to longer average service latencies. Second, GPU and high memory-intensity applications' requests generate a lot of interference by destroying each other's and most importantly latency-sensitive applications' row-buffer locality. With a reasonable maximum batch size (starting from ten onwards), intra-application row-buffer locality is well-preserved with reduced interference to provide good system performance and fairness. We have also observed that most CPU applications rarely form batches that exceed ten requests. This is because the in-order request stream rarely has such a long sequence of requests all to the same row, and the timeout threshold also prevents the batches from becoming too large. As a result, increasing the batch size beyond ten requests does not provide any extra benefit, as shown in Figure 5.9.

DCS FIFO Size. Figure 5.10 shows the sensitivity of SMS to the size of the per-bank FIFOs in the DRAM Command Scheduler (DCS). Fairness degrades as the size of the DCS FIFOs is increased. As the size of the per-bank DCS FIFOs increases, the batch scheduler tends to move

more batches from the batch formation stage to the DCS FIFOs. Once batches are moved to the DCS FIFOs, they cannot be reordered anymore. So even if a higher-priority batch were to become ready, the batch scheduler cannot move it ahead of any batches already in the DCS. On the other hand, if these batches were left in the batch formation stage, the batch scheduler could still reorder them. Overall, it is better to employ smaller per-bank DCS FIFOs that leave more batches in the batch formation stage, enabling the batch scheduler to see more batches and make better batch scheduling decisions, thereby reducing starvation and improving fairness. The FIFOs only need to be large enough to keep the DRAM banks busy.

5.5.4. Case Studies

In this section, we study some additional workload setups and design choices. In view of simulation bandwidth and time constraints, we reduce the simulation time to 200M cycles for these studies.

Case study 1: CPU-only Results. In the previous sections, we showed that SMS effectively mitigates inter-application interference in a CPU-GPU integrated system. In this case study, we evaluate the performance of SMS in a CPU-only scenario. Figure 5.11 shows the system performance and fairness of SMS on a 16-CPU system with exactly the same system parameters as described in Section 5.3.5, except that the system does not have a GPU. We present results only for workload categories with at least some high memory-intensity applications, as the performance/fairness of the other workload categories are quite similar to TCM. We observe that SMS degrades performance by only 4% compared to TCM, while it improves fairness by 25.7% compared to TCM on average across workloads in the "H" category. SMS's performance degradation mainly comes from the "H" workload category (only high memory-intensity applications); as discussed in our main evaluations in Section 4.5, TCM mis-classifies some high memory-intensity applications into the low memory-intensity cluster, starving requests of applications classified into the high memory-intensity cluster. Therefore, TCM gains performance at the cost of fairness. On the other hand, SMS prevents this starvation/unfairness with its probabilistic round-robin policy,

while still maintaining good system performance.

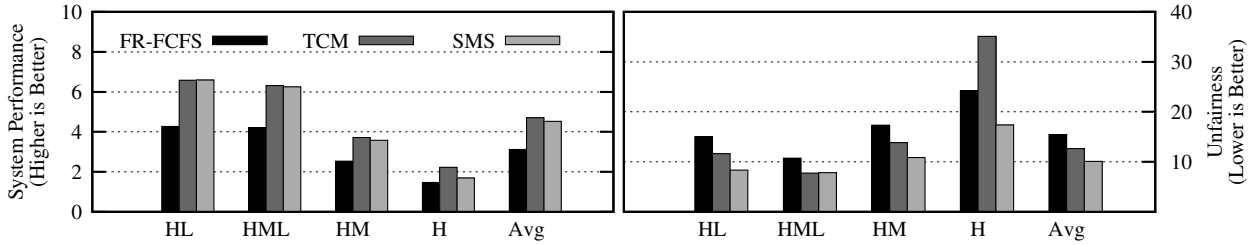


Figure 5.11. System performance and fairness on a 16 CPU-only system.

Case study 2: Always Prioritizing CPU Requests over GPU Requests. Our results in the previous sections show that SMS achieves its system performance and fairness gains by appropriately managing the GPU request stream. In this case study, we consider modifying previously proposed policies by always deprioritizing the GPU. Specifically, we implement variants of the FR-FCFS and TCM scheduling policies, CFR-FCFS and CTCM, where the CPU applications' requests are always selected over the GPU's requests. Figure 5.12 shows the performance and fairness of FR-FCFS, CFR-FCFS, TCM, CTCM and SMS scheduling policies, averaged across workload categories containing high-intensity applications. Several conclusions are in order. First, by protecting the CPU applications' requests from the GPU's interference, CFR-FCFS improves system performance by 42.8% and fairness by 4.82x as compared to FR-FCFS. This is because the baseline FR-FCFS is completely application-unaware and it always prioritizes the row-buffer hitting requests of the GPU, starving other applications' requests. Second, CTCM does not improve system performance and fairness much compared to TCM, because baseline TCM is already application-aware. Finally, SMS still provides much better system performance and fairness than CFR-FCFS and CTCM because it deprioritizes the GPU appropriately, but not completely, while preserving the row-buffer locality within the GPU's request stream. Therefore, we conclude that SMS provides better system performance and fairness than merely prioritizing CPU requests over GPU requests.

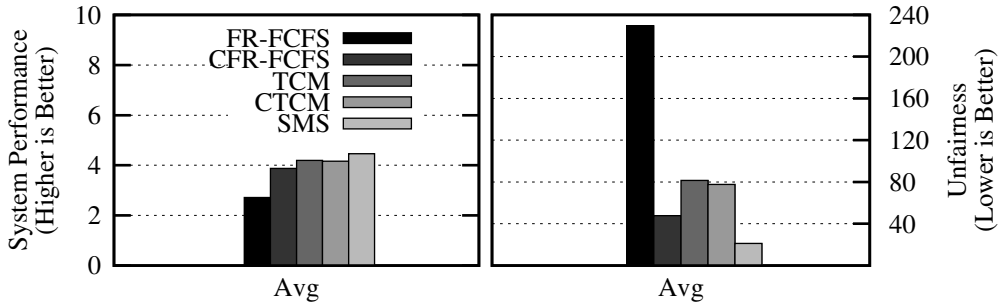


Figure 5.12. Performance and Fairness when always prioritizing CPU requests over GPU requests

5.6. SMS: Conclusion

While many advancements in memory scheduling policies have been made to deal with multi-core processors, the integration of GPUs on the same chip as the CPUs has created new system design challenges. This work has demonstrated how the inclusion of GPU memory traffic can cause severe difficulties for existing memory controller designs in terms of performance and especially fairness. In this dissertation, we propose a new approach, Staged Memory Scheduler, that delivers superior performance and fairness compared to state-of-the-art memory schedulers, while providing a design that is significantly simpler to implement. The key insight behind SMS's scalability is that the primary functions of sophisticated memory controller algorithms can be decoupled, leading to our multi-stage architecture. This research attacks a critical component of a fused CPU-GPU system's memory hierarchy design, but there remain many other problems that warrant further research. For the memory controller specifically, additional explorations will be needed to consider interactions with GPGPU workloads. Co-design and concerted optimization of the cache hierarchy organization, cache partitioning, prefetching algorithms, memory channel partitioning, and the memory controller are likely needed to fully exploit future heterogeneous computing systems, but significant research effort will be needed to find effective, practical, and innovative solutions.

Chapter 6

Reducing Inter-address-space Interference with A TLB-aware Memory Hierarchy

Graphics Processing Units (GPUs) provide high throughput by exploiting a high degree of thread-level parallelism. A GPU executes a group of threads (i.e., a *warp*) in lockstep (i.e., each thread in the warp executes the same instruction concurrently). When a warp stalls, the GPU hides the latency of this stall by scheduling and executing another warp. The use of GPUs to accelerate general-purpose GPU (GPGPU) applications has become common practice, in large part due to the large performance improvements that GPUs provide for applications in diverse domains [56, 68, 145, 351]. The compute density of GPUs continues to grow, with GPUs expected to provide as many as 128 streaming multiprocessors per chip in the near future [32,376]. While the increased compute density can help many GPGPU applications, it exacerbates the growing need to *share* the GPU streaming multiprocessors across multiple applications. This is especially true in large-scale computing environments, such as cloud servers, where a diverse range of application requirements exists. In order to enable efficient GPU hardware utilization in the face of application heterogeneity, these large-scale environments rely on the ability to virtualize the compute resources and execute multiple applications concurrently [6, 10, 160, 165].

The adoption of discrete GPUs in large-scale computing environments is hindered by the prim-

itive virtualization support in contemporary GPUs. While hardware virtualization support has improved for integrated GPUs [54], the current virtualization support for discrete GPUs is insufficient, even though discrete GPUs provide the highest available compute density and remain the platform of choice in many domains [3]. Two alternatives for discrete virtualization are time multiplexing and spatial multiplexing. Emerging GPU architectures support time multiplexing the GPU by providing application preemption [226, 276], but this support currently does not scale well with the number of applications. Each additional application introduces a high degree of contention for the GPU resources (Section 6.1.1). Spatial multiplexing allows us to share a GPU among applications much as we currently share CPUs, by providing support for *multi-address-space concurrency* (i.e., the concurrent execution of kernels from different processes or guest VMs). By efficiently and dynamically managing the kernels that execute concurrently on the GPU, spatial multiplexing avoids the scaling issues of time multiplexing. To support spatial multiplexing, GPUs must provide architectural support for memory virtualization and memory protection domains.

The architectural support for spatial multiplexing in contemporary GPUs is not well-suited for concurrent multi-application execution. Recent efforts at improving address translation support within GPUs [73, 302, 303, 375, 405] eschew MMU-based or IOMMU-based [6, 30] address translation in favor of TLBs close to shader cores. These works do not explicitly target concurrent multi-application execution *within the GPU*, and are instead focused on unifying the CPU and GPU memory address spaces [13]. We perform a thorough analysis of concurrent multi-application execution when these state-of-the-art address translation techniques are employed within a state-of-the-art GPU (Section 6.3). We make four *key observations* from our analysis. First, we find that for concurrent multi-application execution, a shared L2 TLB is more effective than the highly-threaded page table walker and page walk cache proposed in [303] for the unified CPU-GPU memory address space. Second, for both the shared L2 TLB and the page walk cache, TLB misses become a major performance bottleneck with concurrent multi-application execution, despite the latency-hiding properties of the GPU. A TLB miss incurs a high latency, as each miss must walk through multiple levels of a page table to find the desired address translation. Third, we observe that a

single TLB miss can frequently stall multiple warps at once. Fourth, we observe that contention between applications induces significant thrashing on the shared TLB and significant interference between TLB misses and data requests throughout the GPU memory system. Thus, with only a few simultaneous TLB misses, it becomes difficult for the GPU to find a warp that can be scheduled for execution, defeating the GPU's basic techniques for hiding the latency of stalls.

Thus, based on our extensive analysis, we conclude that *address translation becomes a first-order performance concern* in GPUs when multiple applications are executed concurrently. *Our goal* in this work is to develop new techniques that can alleviate the severe address translation bottleneck existing in state-of-the-art GPUs.

To this end, we propose **M**ulti-**A**ddress **S**pace Concurrent **K**ernels (MASK), a new cooperative resource management and TLB design for GPUs that minimizes inter-application interference and translation overheads. MASK takes advantage of locality across shader cores to reduce TLB misses, and relies on three novel techniques to minimize translation overheads. The overarching key idea is to make the entire memory hierarchy *TLB request aware*. First, TLB-FILL TOKENS provide a TLB-selective-fill mechanism to reduce thrashing in the shared L2 TLB, including a bypass cache to increase the TLB hit rate. Second, a low-cost scheme for selectively bypassing TLB-related requests at the L2 cache reduces interference between TLB-miss and data requests. Third, MASK's memory scheduler prioritizes TLB-related requests to accelerate page table walks.

The techniques employed by MASK are highly effective at alleviating the address translation bottleneck. Through the use of TLB-request-aware policies throughout the memory hierarchy, MASK ensures that the first two levels of the page table walk during a TLB miss are serviced quickly. This reduces the overall latency of a TLB miss significantly. Combined with a significant reduction in TLB misses, MASK allows the GPU to successfully hide the latency of the TLB miss through thread-level parallelism. As a result, MASK improves system throughput by 45.7%, improves IPC throughput by 43.4%, and reduces unfairness by 22.4% over a state-of-the-art GPU memory management unit (MMU) design [303]. MASK provides performance within only 23% of a perfect TLB that always hits.

This dissertation makes the following contributions:

- To our knowledge, this is the first work to provide a thorough analysis of GPU memory virtualization under multi-address-space concurrency, and to demonstrate the large impact address translation has on latency hiding within a GPU. We demonstrate a need for new techniques to alleviate interference induced by multi-application execution.
- We design an MMU that is optimized for GPUs that are dynamically partitioned spatially across protection domains, rather than GPUs that are time-shared.
- We propose MASK, which consists of three novel techniques that increase TLB request awareness across the entire memory hierarchy. These techniques work together to significantly improve system performance, IPC throughput, and fairness over a state-of-the-art GPU MMU.

6.1. Background

There has been an emerging need to *share* the GPU hardware among multiple applications. As a result, recent work has enabled support for GPU virtualization, where a single physical GPU can be shared transparently across multiple applications, with each application having its own address space.[1] Much of this work has relied on traditional time multiplexing and spatial multiplexing techniques that have been employed by CPUs, and state-of-the-art GPUs currently contain elements of both types of techniques [361, 368, 381]. Unfortunately, as we discuss in this section, existing GPU virtualization implementations are too coarse, bake fixed policy into hardware, or leave system software without the fine-grained resource management primitives needed to implement truly transparent device virtualization.

[1] In this dissertation, we use the term *address space* to refer to distinct *memory protection domains*, whose access to resources must be isolated and protected during GPU virtualization.

6.1.1. Time Multiplexing

Most modern systems time-share GPUs [226, 272]. These designs are optimized for the case where *no concurrency exists* between kernels from different address spaces. This simplifies memory protection and scheduling at the cost of two fundamental tradeoffs. First, it results in underutilization when kernels from a single address space are unable to fully utilize all of the GPU's resources [174, 189, 191, 284, 382]. Second, it limits the ability of a scheduler to provide forward-progress or QoS guarantees, leaving applications vulnerable to unfairness and starvation [321].

While preemption support could allow a time-sharing scheduler to avoid pathological unfairness (e.g., by context switching at a fine granularity), GPU preemption support remains an active research area [118, 364]. Software approaches [382] sacrifice memory protection. NVIDIA's Kepler [272] and Pascal [276] architectures support preemption at thread block and instruction granularity respectively. We find empirically, that neither is well optimized for inter-application interference.

Figure 6.1 shows the overhead (i.e., performance loss) *per process* when the NVIDIA K40 and GTX 1080 GPUs are contended by multiple application processes. Each process runs a kernel that interleaves basic arithmetic operations with loads and stores into shared and global memory, with interference from a GPU matrix-multiply program. The overheads range from 8% per process for the K40, to 10% or 12% for the GTX 1080. While the performance cost is significant, we also found inter-application interference pathologies to be easy to create: for example, a kernel from one process consuming the majority of shared memory can easily cause kernels from other processes to fail at dispatch. While we expect preemption support to improve in future hardware, we seek a solution that does not depend on it.

6.1.2. Spatial Multiplexing

Resource utilization can be improved with *spatial multiplexing* [4], as the ability to execute multiple kernels *concurrently* enables the system to co-schedule kernels that have complementary resource demands, and can enable independent progress guarantees for different kernels.

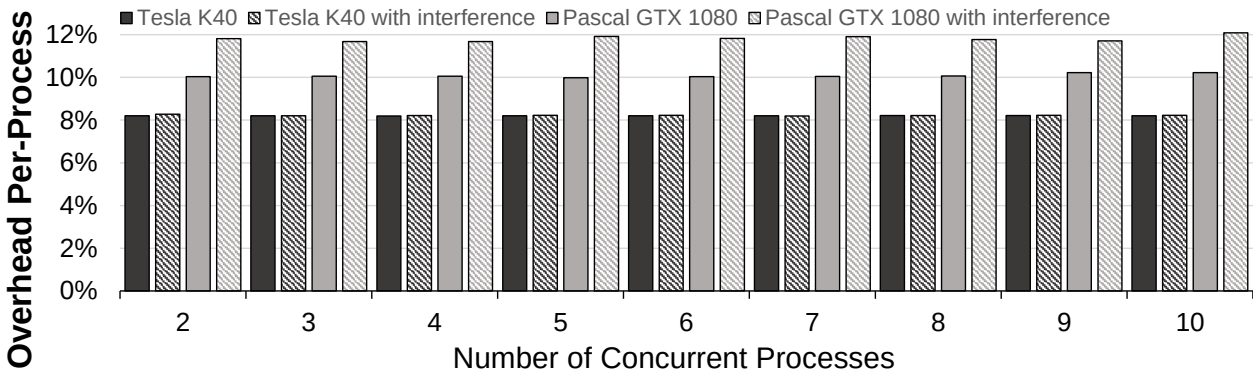


Figure 6.1. Context switch overheads under contention on K40 and GTX 1080.

NVIDIA's stream [272] support, which co-schedules kernels from independent "streams" in a single address space, relies on similar basic concepts, as does application-specific software scheduling of multiple kernels in hardware [174,284] and GPU simulators [33,189,191]. Software approaches (e.g., Elastic Kernels [284]) require programmers to manually time-slice kernels to enable mapping them onto CUDA streams for concurrency. While sharing techniques that leverage the stream abstraction support flexible demand-partitioning of resources, they all share critical drawbacks. When kernels from different applications have complementary resource demands, the GPU remains underutilized. More importantly, merging kernels into a single address space sacrifices memory protection, a key requirement in virtualized settings.

Multi-address-space concurrency support can address these shortcomings by enabling a scheduler to look beyond kernels from the current address space when resources are under-utilized. Moreover, QoS and forward-progress guarantees can be enabled by giving partitions of the hardware simultaneously to kernels from different address spaces. For example, long-running kernels from one application need not complete before kernels from another may be dispatched. NVIDIA and AMD both offer products [9, 147] with hardware virtualization support for statically partitioning GPUs across VMs, but even this approach has critical shortcomings. The system must select from a handful of different partitioning schemes, determined at startup, which is fundamentally inflexible. The system cannot adapt to changes in demand or mitigate interference, which are key goals of virtualization layers.

6.2. Baseline Design

Our goal in this work is to develop efficient address translation techniques for GPUs that allow for flexible, fine-grained spatial multiplexing of the GPU across multiple address spaces, ensuring protection across memory protection domains. Our primary focus is on optimizing a memory hierarchy design extended with TLBs, which are used in a state-of-the-art GPU [303]. Kernels running concurrently on different compute units share components of the memory hierarchy such as lower level caches, so ameliorating contention for those components is an important concern. In this section, we explore the performance costs and bottlenecks induced by different components in a baseline design for GPU address translation, motivating the need for MASK.

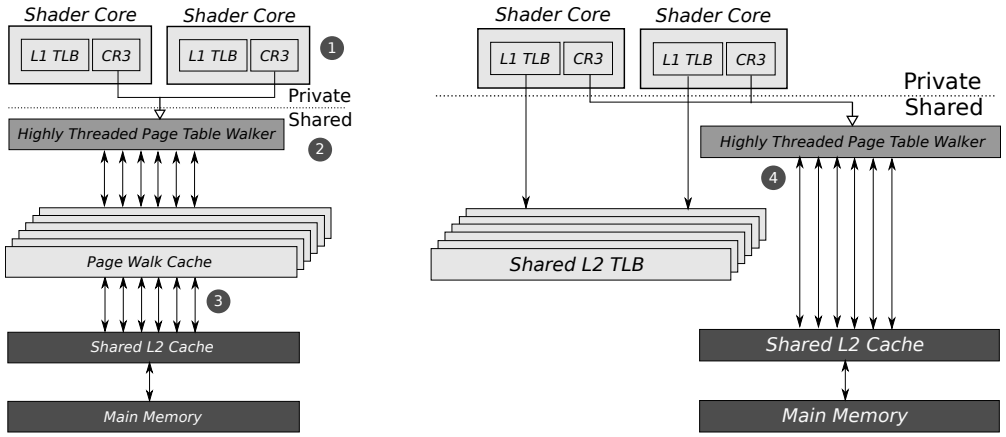
6.2.1. Memory Protection Support

To ensure that memory accesses from kernels running in different address spaces remain isolated, we make use of TLBs and mechanisms for reducing TLB miss costs within the GPU. We adopt the current state-of-the-art for GPU TLB design for CPU-GPU heterogeneous systems proposed by Power et al. [303], and extend the design to handle multi-address-space concurrency, as shown in Figure 6.2a. Each core has a private L1 cache (❶), and all cores share a highly-threaded page table walker (❷). On a TLB miss, the shared page table walker first probes a page walk cache (❸).[2] A miss in the page walk cache goes to the shared L2 cache and (if need be) main memory.

6.2.2. Page Walk Caches

Techniques to avoid misses and hide or reduce their latency are well-studied in the literature. To conserve space, we do not discuss the combinations of techniques that we considered, and focus on the design which we ultimately selected as the baseline for MASK, shown in Figure 6.2b. The design differs from [303] by eliminating the page walk cache, and instead dedicating the same chip area to 1) a shared L2 TLB with entries extended with address space identifiers (ASIDs) and 2) a parallel page table walker. TLB accesses from multiple threads to the same page are coalesced.

[2] In our evaluation, we provision the page walk cache to be 16-way, with 1024 entries.



(a) TLB design from [303] **(b)** MASK's baseline TLB design.

Figure 6.2. Baseline TLB designs. (a) shows the baseline proposed by [303]. (b) shows the baseline used in this dissertation.

On a private L1 TLB miss, the shared L2 TLB is probed (❹). On a shared L2 TLB miss, the page table walker begins a walk, probing the shared L2 cache and main memory.

Figure 6.3 compares the performance with multi-address-space concurrency of our chosen baseline and the design from [303] against the ideal scenario where every TLB access is a hit (see Section 6.5 for our methodology). While Power et al. find that a page walk cache is more effective than a shared L2 TLB [303], the design with a shared L2 TLB provides better performance for all but three workloads, with *13.8% better performance on average*. The shared L2 data cache enables a hit rate for page table walks that is competitive with a dedicated page walk cache, and a shared TLB is a more effective use of chip area. Hence, we adopt a design with a shared L2 TLB as the baseline for MASK. We observe that a 128-entry TLB provides only a 10% reduction in miss rate over a 64-entry TLB, suggesting that the additional area needed to double the TLB size is not efficiently utilized. Thus, we opt for a smaller 64-entry L1 TLB in our baseline. Note that a shared L2 TLB outperforms the page walk cache for both L1 TLB sizes. Lastly, we find that *both designs incur a significant performance overhead compared to the ideal case where every TLB access is a TLB hit.*

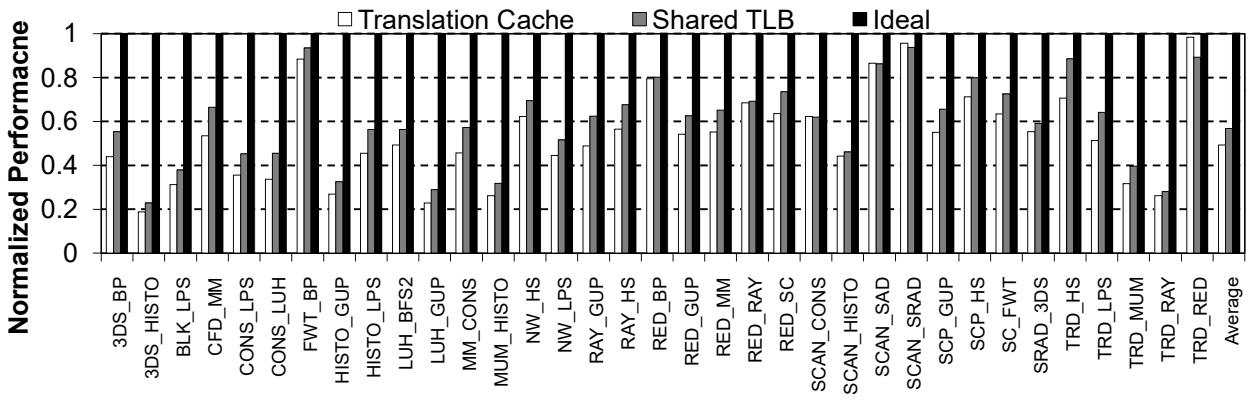


Figure 6.3. Baseline designs vs. ideal performance with no address translation overhead.

6.3. Design Space Analysis

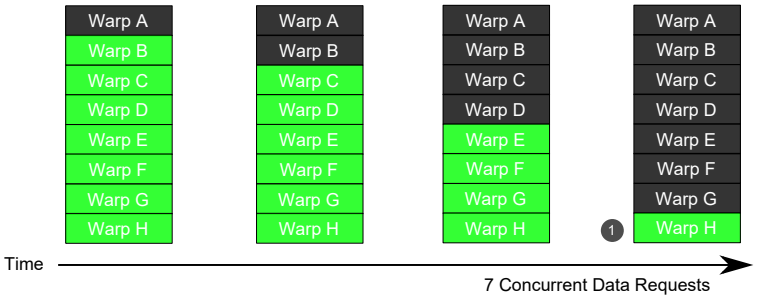
To inform the design of MASK, we characterize overheads for address translation, and consider performance challenges induced by the introduction of multi-address-space concurrency and contention.

6.3.1. Address Translation Overheads

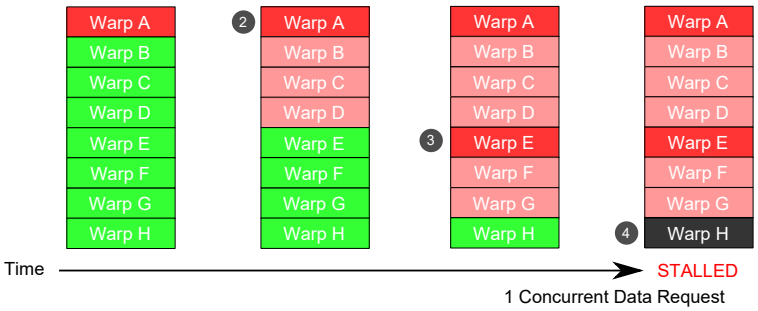
GPU throughput relies on *fine-grained multithreading* [345,365] to hide memory latency. However, we observe a fundamental tension between address translation and fine-grained multithreading. The need to cache address translations at a page granularity, combined with application-level spatial locality, increases the likelihood that translations fetched in response to a TLB miss will be needed by more than one thread. Even with the massive levels of parallelism supported by GPUs, we observe that a small number of outstanding TLB misses can result in the thread scheduler not having enough ready threads to schedule, which in turn limits the GPU's most essential latency-hiding mechanism.

Figure 6.4 illustrates a scenario where all warps of an application access memory. Each box represents a memory instruction, labeled with the issuing warp. Figure 6.4a shows how the GPU behaves when no virtual-to-physical address translation is required. When Warp A executes a high-latency memory access, the core does not stall as long as other warps have schedulable instructions: in this case, the GPU core selects from among the remaining warps (Warps B–H) during

the next cycle (❶), and continues issuing instructions until all requests to DRAM have been sent. Figure 6.4b considers the same scenario *when address translation is required*. Warp A misses in the TLB (indicated in red), and stalls until the translation is fetched from memory. If threads belonging to Warps B–D access data from the same page as the one requested by Warp A, these warps stall as well (shown in light red) and perform no useful work (❷). If a TLB miss from Warp E similarly stalls Warps E–G (❸), only Warp H executes an actual data access (❹). Two phenomena harm performance in this scenario. First, warps that stalled on TLB misses reduce the availability of schedulable warps, lowering utilization. Second, TLB miss requests must complete before actual the data requests can issue, which reduces the ability of the GPU to hide latency by keeping multiple memory requests in flight.



(a) No virtual-to-physical address translation on critical path



(b) Virtual-to-physical address translation on the critical path

Figure 6.4. Example bottlenecks created by address translation. (a) shows the timeline of the execution of GPU warps when no address translation is applied. (b) shows the timeline of the execution of GPU warps when address translation causes a single TLB miss.

Figure 6.5 shows the number of stalled warps per active TLB miss, and the average number of maximum concurrent page table walks (sampled every 10K cycles for a range of applications). In the worst case, a single TLB miss stalls over 30 warps, and over 50 outstanding TLB misses

contend for access to address translation structures. A large number of concurrent misses stall a large number of warps, which must wait before issuing DRAM requests, so minimizing TLB misses and page table walk latency is critical.

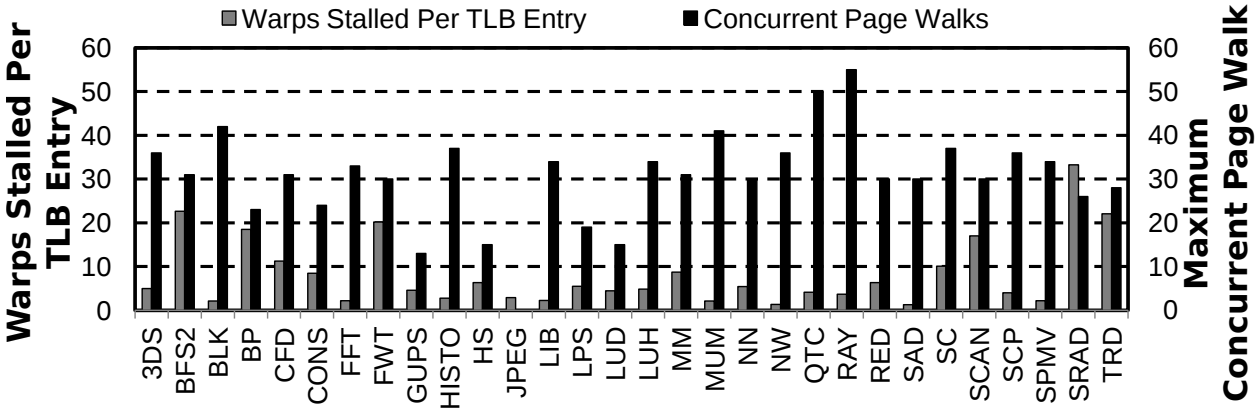


Figure 6.5. The average number of stalled warp per active TLB miss and number of concurrent page walks.

The impact of Large Pages. Larger page size can significantly improve the coverage of the TLB. However, previous work has observed that the use of large pages significantly increases the overhead of demand paging in GPUs [405]. We evaluate this overhead as well as provide a low-overhead multi-page-size support in Chapter 7.

6.3.2. Interference Induced by Resource Sharing

To understand the impact of inter-address-space interference through the memory hierarchy, we concurrently run two applications using the methodology described in Section 6.5. Figure 6.6 shows the TLB miss breakdown across all workloads: most applications incur significant L1 and L2 TLB misses. Figure 6.7 compares the TLB miss rate for applications running in isolation to the miss rates under contention. The data show that inter-address-space interference through additional thrashing has a first-order performance impact.

Figure 6.8 illustrates TLB misses in a scenario where two applications (green and blue) share the GPU. In Figure 6.8a, the green application issues five parallel TLB requests, causing the premature eviction of translations for the blue application, increasing its TLB miss rate (Figure 6.8b).

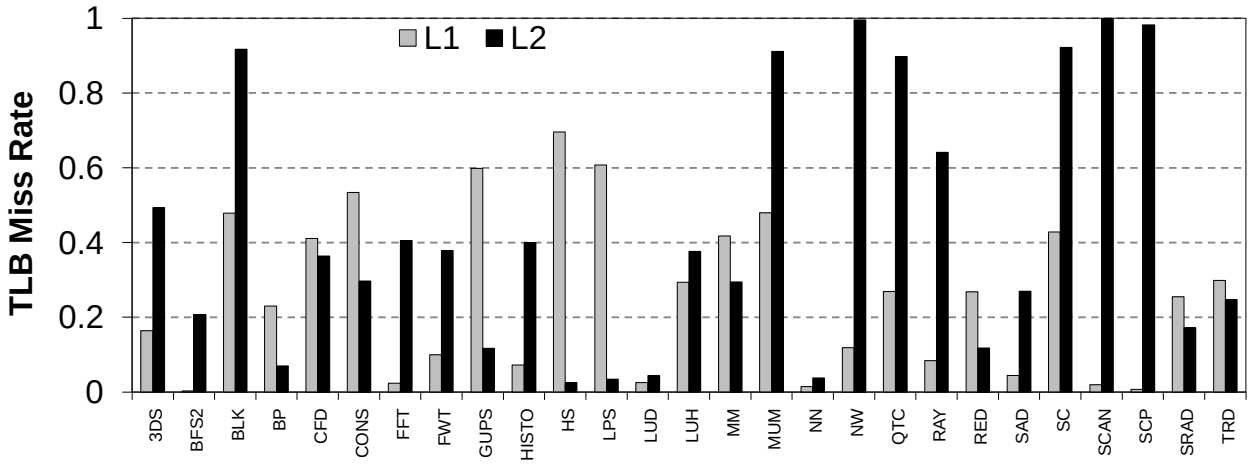


Figure 6.6. TLB miss breakdown for all workloads.

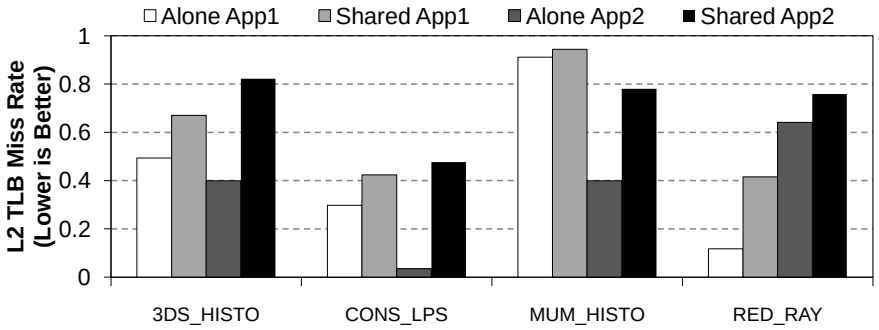
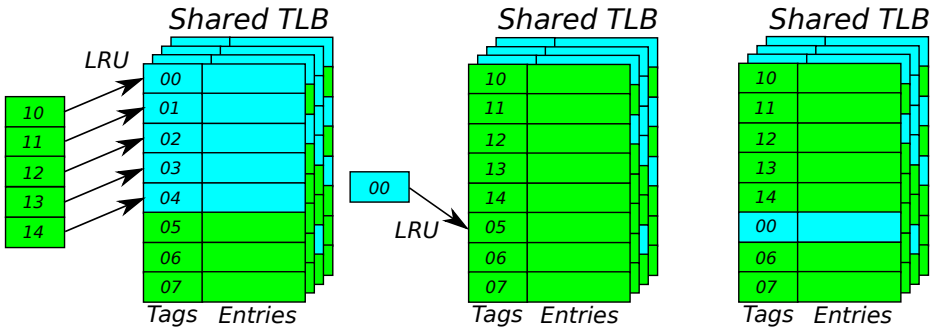


Figure 6.7. Cross-address-space interference in real applications. Each set of bars corresponds to a pair of co-scheduled applications, e.g. "3DS_HISTO" denotes the 3DS and HISTO benchmarks running concurrently.

The use of a shared L2 TLB to cache entries for each application's (non-overlapping) page tables dramatically reduces TLB reach. The resulting inter-core TLB thrashing hurts performance, and can lead to unfairness and starvation when applications generate TLB misses at different rates. Our findings of severe performance penalties for increased TLB misses corroborate previous work on GPU memory designs [302, 303, 375]. However, interference across address spaces can inflate miss rates in ways not addressed by these works, and which are best managed with mechanisms that are aware of concurrency (as we show in Section 6.6.1).



(a) Parallel requests **(b)** Conflict **(c)** Final state

Figure 6.8. Cross-address-space TLB interference. (a) shows parallel TLB accesses generated from green application evicting entries from the blue application. (b) shows the conflict as the blue application TLB entries was evicted earlier, creating a conflict miss (c).

6.3.3. Interference from Address Translation

Interference at the Shared Data Cache. Prior work [36] demonstrated that while cache hits in GPUs reduce the consumption of off-chip memory bandwidth, cache hits result in a lower load/store instruction latency only when *every thread in the warp* hits in the cache. In contrast, when a page table walk hits in the shared L2 cache, the cache hit has the potential to help reduce the latency of *other warps* that have threads which access the same page in memory. While this makes it desirable to allow the data generated by the page table walk to consume entries in the shared cache, TLB-related data can still interfere with and thrash normal data cache entries, which hurts the overall performance.

Hence, a trade-off exists between prioritizing TLB related requests or normal data requests in the GPU memory hierarchy. Figure 6.9 shows that entries for translation data from levels closer to the page table root are more likely to be shared across warps, and will typically be served by cache hits. Allowing shared structures to cache page walk data from only the levels closer to the root could alleviate the interference between low-hit-rate translation data and application data.

Interference at the Main Memory. Figure 6.10 characterizes the DRAM bandwidth utilization, broken down between data and address translation requests for applications sharing the GPU concurrently pairwise. Figure 6.11 compares the average latency for data requests and translation requests. We see that even though page walk requests consume only 13.8% of the utilized DRAM

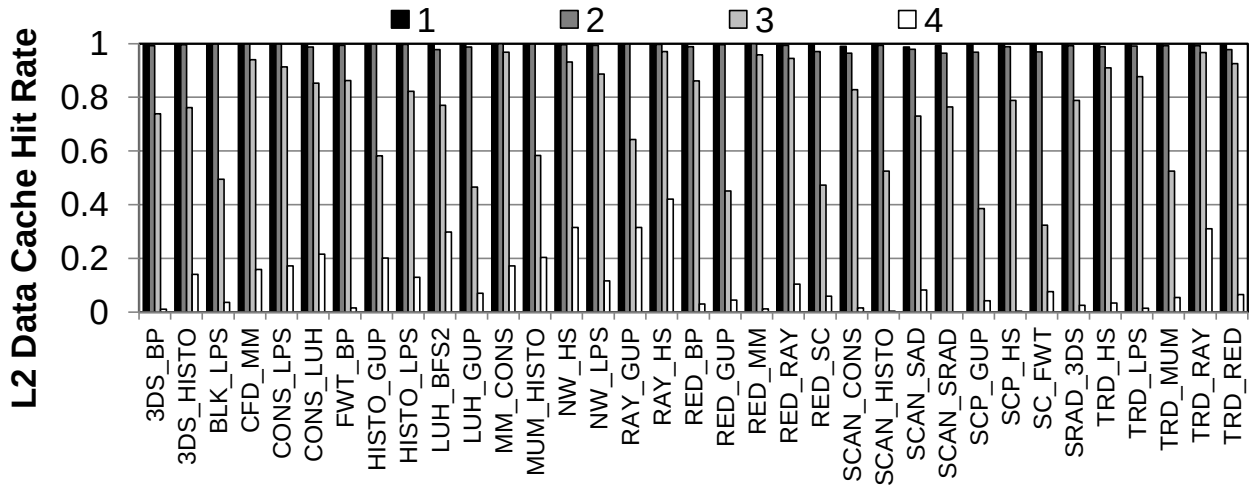


Figure 6.9. L2 cache hit rate for page walk requests.

bandwidth (2.4% of the maximum bandwidth), their DRAM latency is higher than that of data requests, which is particularly egregious since data requests that lead to TLB misses stall while waiting for page walks to complete. The phenomenon is caused by FR-FCFS memory schedulers [317, 406], which prioritize accesses that hit in the row buffer. Data requests from GPGPU applications generally have very high row buffer locality [33, 189, 385, 397], so a scheduler that cannot distinguish page walk requests effectively de-prioritizes them, increasing their latency.

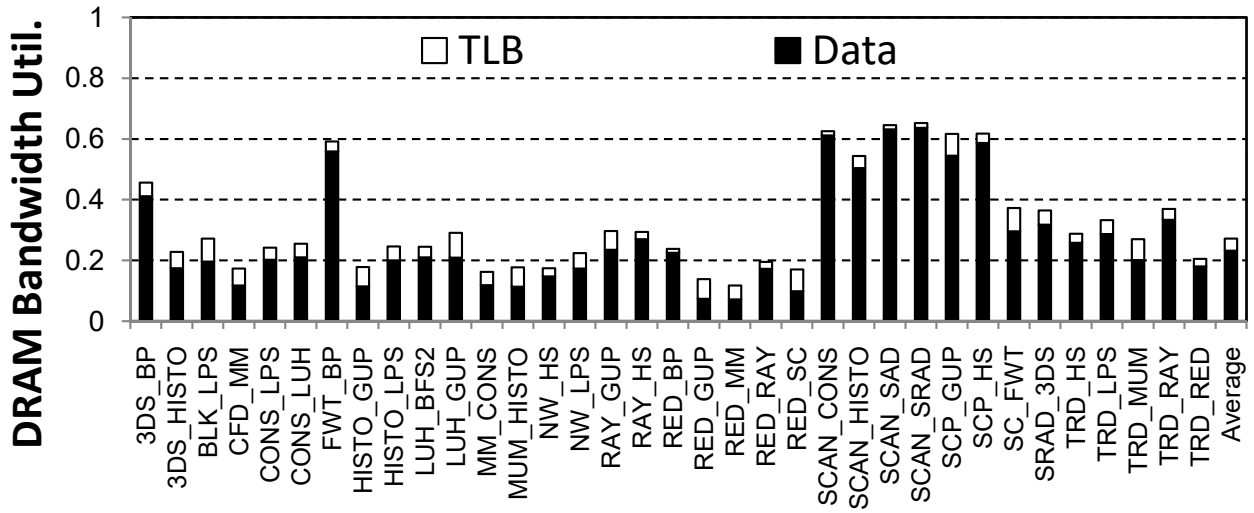


Figure 6.10. Bandwidth breakdown of two applications.

In summary, we make two important observations about address translation in GPUs. First, address translation competes with the GPU's ability to hide latency through thread-level parallelism,

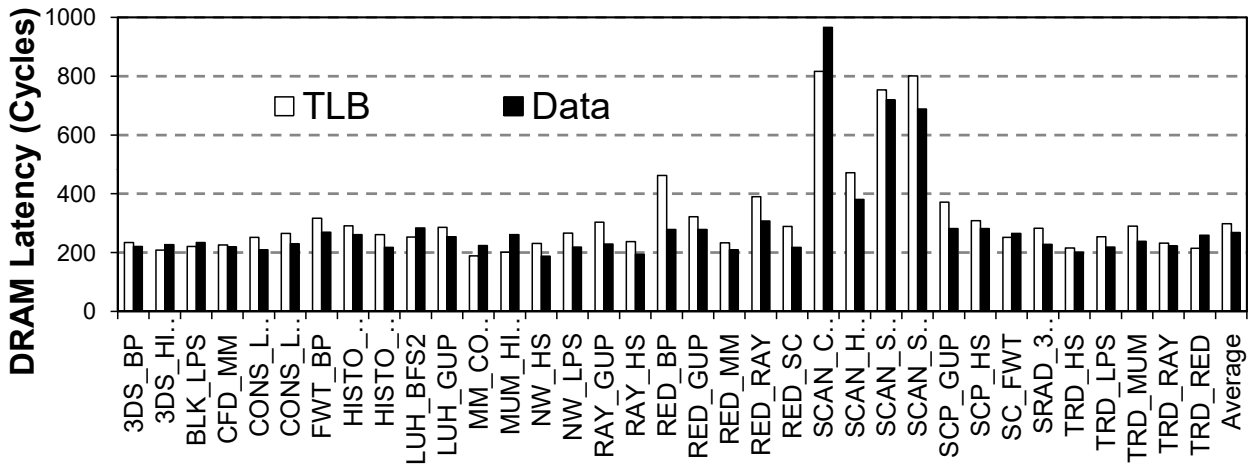


Figure 6.11. Latency breakdown of two applications.

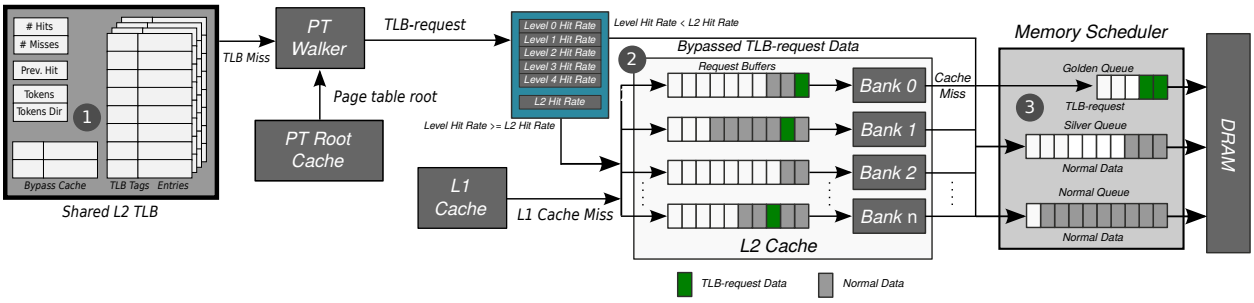


Figure 6.12. MASK design overview: (1) TLB-FILL TOKENS, (2) TLB-REQUEST-AWARE L2 BYPASS and (3) ADDRESS-SPACE-AWARE DRAM SCHEDULER.

when multiple warps stall on the TLB misses for a single translation. Second, the GPU's memory-level parallelism generates interference across address spaces, and between TLB requests and data requests, which can lead to unfairness and increased latency. In light of these observations, *the goal of this work* is to design mechanisms that alleviate the translation overhead by 1) increasing the TLB hit rate through reduced TLB thrashing, 2) decreasing interference between normal data and TLB requests in the shared L2 cache, 3) decreasing TLB miss latency by prioritizing TLB-related requests in DRAM, and 4) enhancing memory scheduling to provide fairness without sacrificing DRAM bandwidth utilization.

6.4. The Design of MASK

We now introduce Multi Address Space Concurrent Kernels (MASK), a new cooperative resource management framework and TLB design for GPUs. Figure 6.12 provides a design overview of MASK. MASK employs three components in the memory hierarchy to reduce address translation overheads while requiring a minimal hardware change. First, we introduce TLB-FILL TOKENS to lower the number of TLB misses and utilize a bypass cache to cache frequently used TLB entries (❶). Second, we design a TLB-REQUEST-AWARE L2 BYPASS mechanism for TLB requests that significantly increases the shared L2 data cache utilization, by reducing interference from TLB misses at the shared L2 data cache (❷). Third, we design an ADDRESS-SPACE-AWARE DRAM SCHEDULER to further reduce interference between TLB requests and data requests from different applications (❸). We analyze the hardware cost of MASK in Section 6.6.4.

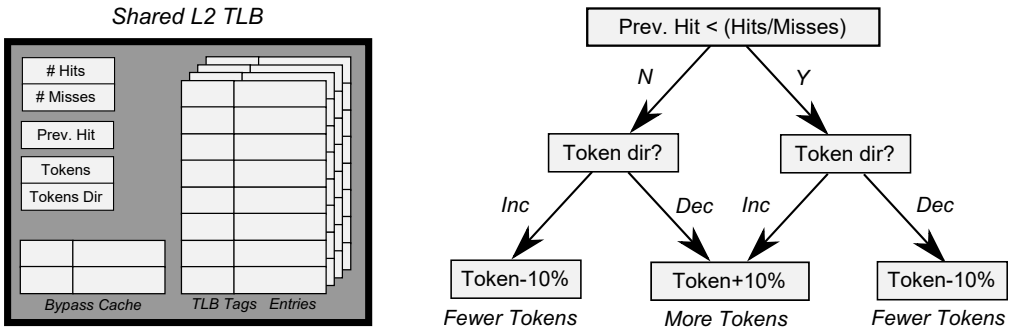
6.4.1. Memory Protection

MASK uses per-core page table root registers (similar to x86 CR3) to set the current address space on each core: setting it also sets the value in a page table root cache with per-core entries at the L2 layer. The page table root cache is kept coherent with the CR3 value in the core by draining all in-flight memory requests for that core when the page table root is set. L2 TLB cache lines are extended with address-space identifiers (ASIDs); TLB flush operations target a single shader core, flushing the core's L1 TLB and all entries in the L2 TLB with a matching ASID.

6.4.2. Reducing L2 TLB Interference

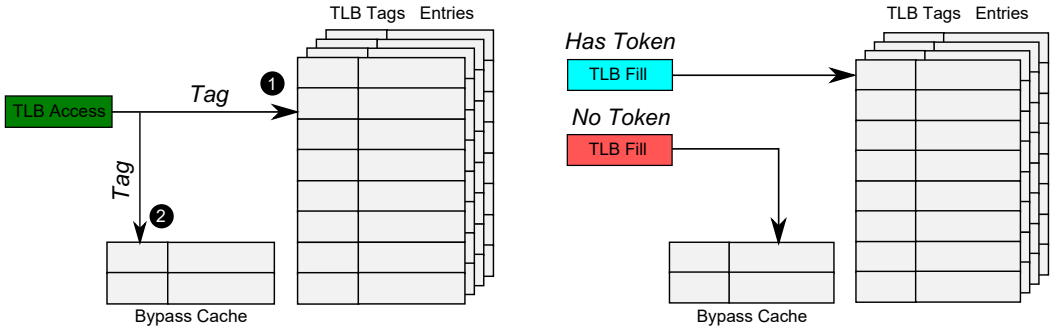
Sections 6.3.1 and 6.3.2 demonstrated the need to minimize TLB miss overheads. MASK addresses this need with a new mechanism called TLB-FILL TOKENS. Figure 6.13a shows architectural additions to support TLB-FILL TOKENS. We add two 16-bit counters to track TLB hits and misses per shader core, along with a small fully-associative bypass cache to the shared TLB. Figure 6.14 illustrates the operation of the proposed TLB fill bypassing logic. When a TLB access arrives (Figure 6.14a), tags for both the shared TLB (❶) and bypass cache (❷) are probed

in parallel. A hit on either the TLB or the bypass cache yields a TLB hit.



(a) TLB hit and miss counters **(b)** TLB counter control logic

Figure 6.13. L2 TLB and token assignment logic. (a) shows the hardware components to support TLB-FILL TOKENS. (b) shows the control logic that determines the number of tokens for TLB-FILL TOKENS.



(a) TLB access **(b)** TLB fill

Figure 6.14. TLB fill bypassing logic in MASK.

To reduce inter-core thrashing at the shared L2 TLB, we use an epoch- and token-based scheme to limit the number of warps from each shader core that can fill into (and therefore contend for) the L2 TLB. While every warp can probe the shared TLB, to prevent thrashing, we allow only warps with tokens to fill into the shared TLB as shown in Figure 6.14b. This token-based mechanism requires two components, one to determine the number of tokens for each application, and one to implement a policy for assigning tokens to warps.

Determining the Number of Tokens. At the beginning of a kernel, MASK performs no bypassing, but tracks the L2 miss rate for each application and the total number of warps in each core. After the first epoch,[3] the initial number of tokens (*InitialToken*) is set to a fraction of the

[3]We empirically select an epoch length of 100K cycles.

total number of warps per application. At the end of any subsequent epoch, MASK compares the shared L2 TLB miss rates of the current and previous epoch. If the miss rate decreases or increases from the previous epoch, MASK uses the logic shown in Figure 6.13b to decrease or increase the number of tokens allocated to each application.

Assigning Tokens to Warps. Empirically, we observe that 1) warps throughout the GPU cores have mostly even TLB miss rate distribution; and 2) it is more beneficial for warps that previously have tokens to retain their token, as it is more likely that their TLB entries are already in the shared TLB. We leverage these two observations to simplify the token assignment logic: TLB-FILL TOKENS simply hands out tokens in a round-robin fashion to all cores in warpID order. The heuristic is effective at reducing thrashing, as contention at the shared TLB is reduced based on the number of tokens, and highly-used TLB entries that do not have tokens can still fill into the bypassed cache.

Bypass Cache. While TLB-FILL TOKENS can reduce thrashing in the shared TLB, a handful of highly-reused pages from warps with no tokens may be unable to utilize the shared TLB. To address this, we add a *bypass cache*, which is a small 32-entry fully associative cache. Only warps without tokens can fill the bypass cache.

Replacement Policy. While it is possible to base the cache replacement policy on how many warps are stalled per TLB entry and prioritize TLB entries with more warps sharing an entry, we observe small variance across TLB entries on the shared TLB in practice. Consequently, a replacement policy based on the number of warps stalled per TLB entry actually performs worse than a reuse-based policy. Hence, we use LRU replacement policy for L1 TLBs, the shared L2 TLB and the bypass cache.

6.4.3. Minimizing Shared L2 Interference

Interference from TLB Requests. While Power et al. propose to coalesce TLB requests to minimize the cache pressure and performance impact [303], we find that a TLB miss generates shared cache accesses with varying degrees of locality. Translating addresses through a multi-

level page table (4 levels for MASK) can generate dependent memory requests for each level. This causes significant queuing latency at the shared L2 cache, corroborating observations from previous work [36]. Page table entries in levels closer to the root are more likely to be shared across threads than entries near the leaves, and more often hit in the shared L2 cache.

To address both the interference and queuing delay at the shared L2 cache we introduce TLB-REQUEST-AWARE L2 BYPASS for TLB requests, as shown in Figure 6.15. To determine which TLB requests should be bypassed, we leverage our insights from Section 6.3.3. Because of the sharp drop-off in the L2 cache hit rate after the first few levels, we can simplify the bypassing logic to compare the L2 cache hit rate of each page level for TLB requests to the L2 cache hit rate for non-TLB requests. We impose L2 cache bypassing when the hit rate for TLB requests falls below the hit rate for non-TLB requests. Memory requests are tagged with three additional bits specifying page walk depth, allowing MASK to differentiate between request types. These bits are set to *zero* for normal data requests, and to 7 for any depth higher than 6.[4]

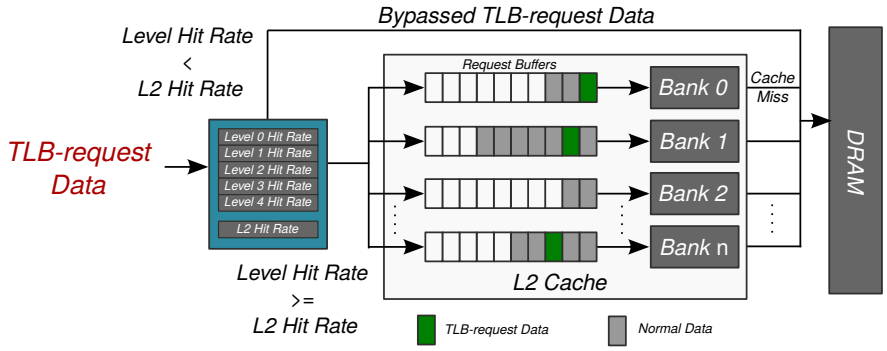


Figure 6.15. The design of TLB-REQUEST-AWARE L2 BYPASS.

6.4.4. Minimizing Interference at Main Memory

Section 6.3.3 demonstrates two different types of interference at the main memory. Normal data requests can interfere with TLB requests, and data requests from multiple applications can interfere with each other. MASK's memory controller design mitigates both forms of interference using an ADDRESS-SPACE-AWARE DRAM SCHEDULER.

[4]Note that all experiments done in this dissertation use a depth of 4.

MASK's ADDRESS-SPACE-AWARE DRAM SCHEDULER breaks the traditional DRAM request buffer into three separate queues, as shown in Figure 6.12. The first queue, called the GOLDEN QUEUE, contains a small FIFO queue.[5] TLB-related requests always go to the GOLDEN QUEUE, while non-TLB-related requests go the other two larger queues (similar to the size of a typical DRAM request buffer size). The second queue, called the SILVER QUEUE, contains data request from *one* selected application. The last queue, called the NORMAL QUEUE, contains data requests from all other applications. The GOLDEN QUEUE is used to prioritize TLB misses over data requests, while the SILVER QUEUE ensures that DRAM bandwidth is fairly distributed across applications.

Our ADDRESS-SPACE-AWARE DRAM SCHEDULER always prioritizes requests in the GOLDEN QUEUE over requests in the SILVER QUEUE, which are prioritized over requests in the NORMAL QUEUE. Applications take turns being assigned to the SILVER QUEUE based on two factors: the number of concurrent page walks, and the number of warps stalled per active TLB miss. The number of requests each application can add to the SILVER QUEUE is shown in Equation 6.1. Application (App_i) inserts $thres_i$ requests into the SILVER QUEUE. Then, the next application (App_{i+1}) is allowed to send $thres_{i+1}$ requests to the SILVER QUEUE. Within each queue, FR-FCFS [317, 406] is used to schedule requests.

$$thres_i = thres_{max} \frac{Concurrent_i * WrpStalled_i}{\sum_{j=1}^{numApp} Concurrent_j * WrpStalled_j} \tag{6.1}$$

To track the number of outstanding concurrent page walks, we add a 6-bit counter per application to the shared TLB.[6] This counter tracks of the *maximum* number of TLB miss queue, and is used as $Concurrent_i$ in Equation 6.1. To track the number of warps stalled per active TLB, we add a 6-bit counter to the TLB MSHRs, to track the maximum number of warps that hit in each MSHR entry. This number is used for $WrpStalled_i$. Note that the ADDRESS-SPACE-AWARE DRAM SCHEDULER resets all of these counters every epoch.[3]

[5] We observe that TLB-related requests have low row locality. Thus, we use a FIFO queue to further simplify the design.

[6] We leave techniques to virtualize this counter for more than 64 applications as future work.

We find that the number of concurrent TLB requests that go to each memory channel is small, so our design has an additional benefit of lowering page table walk latency while minimizing interference. The SILVER QUEUE prevents bandwidth-heavy applications from interfering with applications utilizing this silver queue, which in turn prevents starvation. It also minimizes the reduction in total bandwidth utilization, as the per-queue FR-FCFS scheduling policy ensures that application-level row buffer locality is preserved.

6.4.5. Page Faults and TLB Shootdowns

Address translation inevitably introduces page faults. Our design can be extended to use techniques from previous works, such as performing copy-on-write for minor faults [303], and either exception support [244] or demand paging techniques [15,276,405] for major faults. We leave this as future work, and do not evaluate these overheads.

Similarly, TLB shootdowns are required when shader cores change address spaces and when page tables are updated. We do not envision applications that make frequent changes to memory mappings, so we expect such events to be rare. Techniques to reduce TLB shootdown overhead [52,320] are well-explored and can be applied to MASK.

6.5. Methodology

We model Maxwell architecture [273] cores, TLB fill bypassing, bypass cache, and memory scheduling mechanisms in MASK using the MAFIA framework [174], which is based on GPGPU-Sim 3.2.2 [41]. We heavily modify the simulator to accurately model the behavior of CUDA Unified Virtual Address [273,276] as described below. Table 7.1 provides details on our baseline GPU configuration. In order to show that MASK works on any GPU architecture, we also evaluate the performance of MASK on a Fermi architecture [271], which we discuss in Section 6.6.3.

TLB and Page Table Walk Model. We modify the MAFIA framework to accurately model the TLB designs from [303] and the MASK baseline design. We employ the non-blocking TLB implementation used in the design from Pichai et al. [302]. Each core has a private L1 TLB. The

System Overview	30 cores, 64 execution unit per core. 8 memory partitions
Shader Core Config	1020 MHz, 9-stage pipeline, 64 threads per warp, GTO scheduler [318]
Private L1 Cache	16KB, 4-way associative, LRU, L1 misses are coalesced before accessing L2, 1 cycle latency
Shared L2 Cache	2MB total, 16-way associative, LRU, 2 cache banks 2 interconnect ports per memory partition, 10 cycle latency
Private L1 TLB	64 entries per core, fully associative, LRU, 1 cycle latency
Shared L2 TLB	512 entries total, 16-way associative, LRU, 2 ports per memory partition (16 ports total), 10 cycle latency
DRAM	GDDR5 1674 MHz, 8 channels, 8 banks per rank FR-FCFS scheduler [317, 406] burst length 8
Page Table Walker	64 threads shared page table walker, traversing 4-level page table

Table 6.1. The configuration of the simulated system.

page table walker is shared, and admits up to 64 concurrent threads for walks. The baseline design for MASK adds a shared L2 TLB instead of page walk caches (see Section 6.2.2), with a shared L2 TLB in each memory partition. Both L1 and L2 TLB entries contain MSHR entries to track in-flight page table walks. On a TLB miss, a page table walker generates a series of dependent requests that probe the L2 data cache and main memory as needed. To correctly model virtual-to-physical address mapping and dependent memory accesses for multi-level page walks, we collect traces of all virtual addresses referenced by each application (executing them to completion), enabling us to pre-populate disjoint physical address spaces for each application with valid page tables.

Workloads. We randomly select 27 applications from the CUDA SDK [270], Rodinia [68], Parboil [351], LULESH [185, 186], and SHOC [80] suites. We classify these benchmarks based on their L1 and L2 TLB miss rates into one of four groups. Table 6.2 shows the categorization for each benchmark. For our multi-application secs/mask-micro17/results, we randomly select 35 pairs of applications, avoiding combinations that select applications from the *lowL1miss-lowL2miss* category, as these applications are relatively insensitive to memory protection overheads. The applica-

tion that finishes first is relaunched to properly model contention.

We divide these pairs into three workload categories based on the number of applications that are from *highL1miss-highL2miss* category. *0 HMR* contains workload bundles where *none* of the applications in the bundle are from *highL1miss-highL2miss*. *1 HMR* contains workloads where *only one* application in the bundle is from *highL1miss-highL2miss*. *2 HMR* contains workloads where *both* applications in the bundle are from *highL1miss-highL2miss*.

L1 TLB Miss	L2 TLB Miss	Benchmark Name
Low	Low	LUD, NN
Low	High	BFS2, FFT, HISTO, NW, QTC, RAY, SAD, SCP
High	Low	BP, GUP, HS, LPS
High	High	3DS, BLK, CFD, CONS, FWT, LUH, MM, MUM, RED, SC, SCAN, SRAD, TRD

Table 6.2. Categorization of each benchmark.

Evaluation Metrics. We report performance using weighted speedup [97, 98], defined as $\sum \frac{IPC_{Shared}}{IPC_{Alone}}$. IPC_{alone} is the IPC of an application that runs on the same number of shader cores, but does not share GPU resources with any other applications, and IPC_{shared} is the IPC of an application when running concurrently with other applications. We report the unfairness of each design using maximum slowdown, defined as $Max\frac{IPC_{Alone}}{IPC_{Shared}}$ [33, 94].

Scheduling and Partitioning of Cores. The design space for core scheduling is quite large, and finding optimal algorithms is beyond the scope of this dissertation. To ensure that we model a scheduler that performs reasonably well, we assume an oracle schedule that finds the best partition for each pair of applications. For each pair of applications, concurrent execution partitions the cores according to the best weighted speedup observed for that pair during an exhaustive search over all possible partitionings.

Design Parameters. MASK exposes two configurable parameters: $InitialTokens$ for TLB-FILL TOKENS and $thres_{max}$ for the ADDRESS-SPACE-AWARE DRAM SCHEDULER. A sweep over the range of possible $InitialTokens$ values reveals less than 1% performance variance, as

TLB-FILL TOKENS is effective at reconfiguring the total number of tokens to a steady-state value (shown in Figure 6.14). In our evaluation, we set *InitialTokens* to 80%. We set $thres_{max}$ to 500 empirically.

6.6. Evaluation

We compare the performance of MASK against three designs. The first, called *Static*, uses a static spatial partitioning of resources, where an oracle is used to partition GPU cores, but the shared L2 cache and memory channels are partitioned equally to each application. This design is intended to capture key design aspects of NVIDIA GRID and AMD FirePro—however, insufficient information is publicly available to enable us to build a higher fidelity model. The second design, called *GPU-MMU*, models the flexible spatial partitioning GPU MMU design proposed by Power et al. [303].[7] The third design we compare to is an *ideal* scenario, where every single TLB access is a TLB hit. We also report performance impact for individual components of MASK: TLB-FILL TOKENS (MASK-TLB), TLB-REQUEST-AWARE L2 BYPASS (MASK-Cache), and ADDRESS-SPACE-AWARE DRAM SCHEDULER (MASK-DRAM).

6.6.1. Multiprogrammed Performance

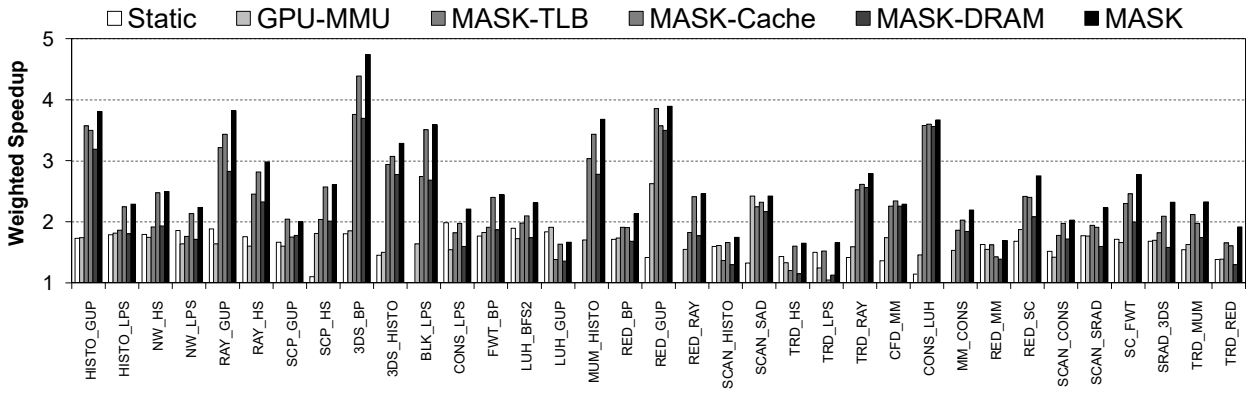


Figure 6.16. System-wide weighted speedup for multiprogrammed workloads.

Figures 6.17 and 6.16 compare the weighted speedup of multiprogrammed workloads for MASK, as well as each of the components of MASK, against Static and GPU-MMU. Each group of

[7]Note that we use the design in Figure 6.2b instead of the one in Figure 6.2a, as it provides better performance for the workloads that we evaluate.

bars in the figure represents a pair of co-scheduled benchmarks. Compared to GPU-MMU, MASK provides 45.7% additional speedup. We also found that MASK performs only 23% worse than the ideal scenario where the TLB always hits. We observe that MASK provides 43.4% better aggregate throughput (system wide IPC) compared to GPU-MMU. Compared to the Static baseline, where resources are statically partitioned, both GPU-MMU and MASK provide better performance, because when an application stalls for concurrent TLB misses, it does not use other shared resources such as DRAM bandwidth. During such stalls, other applications can utilize these resources. When multiple GPGPU applications run concurrently, TLB misses from two or more applications can be staggered, increasing the likelihood that there will be heterogeneous and complementary resource demand.

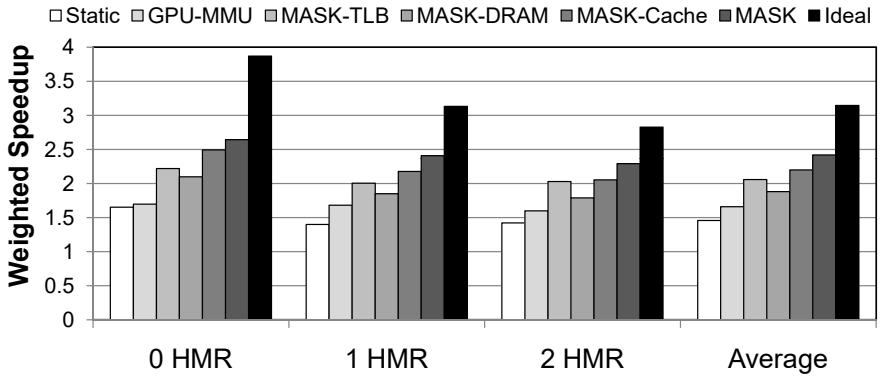


Figure 6.17. System-wide weighted speedup for multiprogrammed workloads.

Figure 6.18 compares unfairness in MASK against the GPU-MMU and Static baselines. On average, our mechanism reduces unfairness by 22.4% compared to GPU-MMU. As the number of tokens for each application changes based on the TLB miss rate, applications that benefit more from the shared TLB are more likely to get more tokens, causing applications that do not benefit from shared TLB space to yield that shared TLB space to other applications. Our application-aware token distribution mechanism and TLB fill bypassing mechanism can work in tandem to reduce the amount of inter-application cache thrashing observed in Section 6.3.2. Compared to statically partitioning resources in Static, allowing both applications access to all of the shared resources provides better fairness. On average, MASK reduces unfairness by 30.7%, and a handful

of applications benefit by as much as 80.3%.

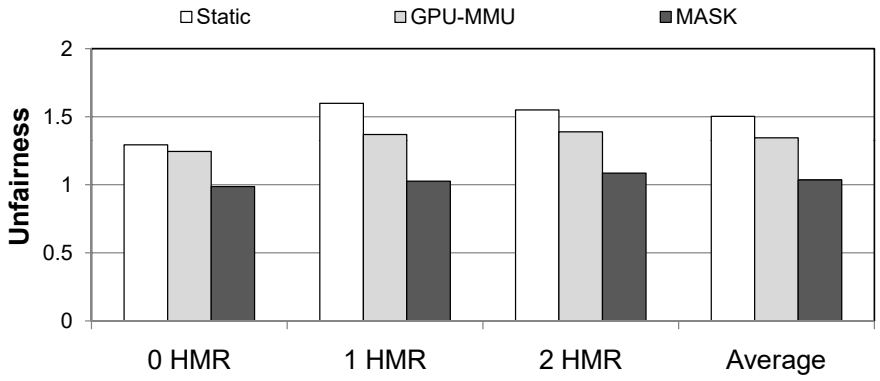


Figure 6.18. Max unfairness of GPU-MMU and MASK.

Individual Application Analysis. MASK provides better throughput on all applications sharing the GPU due to reduced TLB miss rates for each application. The per-application L2 TLB miss rates are reduced by over 50% on average, which is in line with the system-wide miss rates observed in Figure 6.3. Reducing the number of TLB misses through the TLB fill bypassing policy (Section 6.4.2), and reducing the latency of TLB misses through the shared L2 bypassing (Section 6.4.3) and the TLB- and application-aware DRAM scheduling policy (Section 6.4.4) enables significant performance improvement.

In some cases, running two applications concurrently provides better speedup than running the application alone (e.g., RED-BP, RED-RAY, SC-FWT). We attribute these cases to substantial improvements (more than 10%) of two factors: a lower L2 queuing latency for bypassed TLB requests, and a higher L1 hit rate when applications share the L2 and main memory with other applications.

6.6.2. Component-by-Component Analysis

***The Effectiveness of* TLB-FILL TOKENS.** Table 6.3 compares the TLB hit rates of GPU-MMU and MASK-TLB. We show only GPU-MMU results for TLB hit rate experiments, as the TLB hit behavior for Static and GPU-MMU are similar. MASK-TLB increases TLB hit rates by 49.9% on average, which we attribute to TLB-FILL TOKENS. First, TLB-FILL TOKENS reduces the number of warps utilizing the shared TLB entries, which in turn reduces the miss rate. Second,

the bypass cache can store frequently-used TLB entries that cannot be filled in the traditional TLB. Table 6.4 confirms this, showing the hit rate of the bypass cache for MASK-TLB. From Table 6.3 and Table 6.4, we conclude that the TLB-fill bypassing component of MASK successfully reduces thrashing and ensures that frequently-used TLB entries stay cached.

Shared TLB Hit Rate	0 HMR	1 HMR	2 HMR	Average
GPU-MMU	47.8%	45.6%	55.8%	49.3%
MASK-TLB	68.1%	75.2%	76.1%	73.9%

Table 6.3. Aggregate Shared TLB hit rates.

Bypass Cache Hit Rate	0 HMR	1 HMR	2 HMR	Average
MASK-TLB	63.9%	66.6%	68.8%	66.7%

Table 6.4. TLB hit rate for bypassed cache.

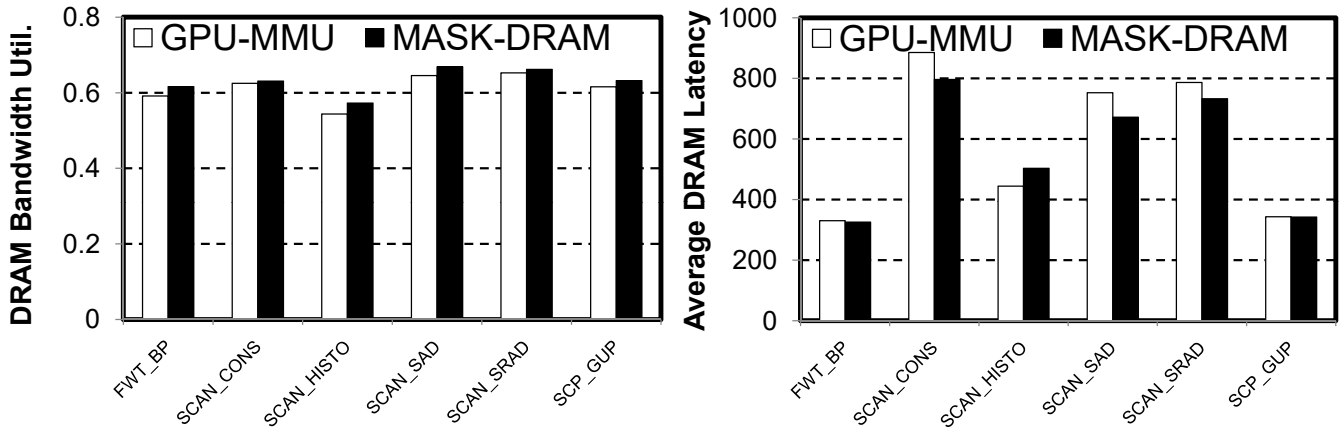
***The Effectiveness of* TLB-REQUEST-AWARE L2 BYPASS.** Table 6.5 shows the average L2 data cache hit rate for TLB requests. For requests that fill into the shared L2 data cache, TLB-REQUEST-AWARE L2 BYPASS is effective in selecting which blocks to cache, resulting in a TLB request hit rate that is higher than 99% for all of our workloads. At the same time, TLB-REQUEST-AWARE L2 BYPASS minimizes the impact of bypassed TLB requests, leading to 17.6% better performance on average compared to GPU-MMU, as shown in Figure 6.17.

L2 Data Cache Hit Rate	0 HMR	1 HMR	2 HMR	Average
GPU-MMU	71.7%	71.6%	68.7%	70.7%
MASK-Cache	97.9%	98.1%	98.8%	98.3%

Table 6.5. L2 data cache hit rate for TLB requests.

***Effectiveness of* ADDRESS-SPACE-AWARE DRAM SCHEDULER.** While the impact of the DRAM scheduler we propose is minimal for many applications, (the average improvement across

all workloads is just 0.83% in Figure 6.17), we observe that a few applications that suffered more severely from interference (see Figures 6.10 and 6.11) can significantly benefit from our scheduler, since it prioritizes TLB-related requests. Figures 6.19a and 6.19b compare the DRAM bandwidth utilization and DRAM latency of GPU-MMU and MASK-DRAM for workloads that benefit from ADDRESS-SPACE-AWARE DRAM SCHEDULER. When our DRAM scheduler policy is employed, SRAD from the SCAN-SRAD pair sees an 18.7% performance improvement, while both SCAN and CONS from SCAN-CONS have performance gains of 8.9% and 30.2%, respectively. In cases where the DRAM latency is high, the DRAM scheduler policy reduces the latency of TLB requests by up to 10.6% (SCAN-SAD), while increasing DRAM bandwidth utilization by up to 5.6% (SCAN-HISTO).



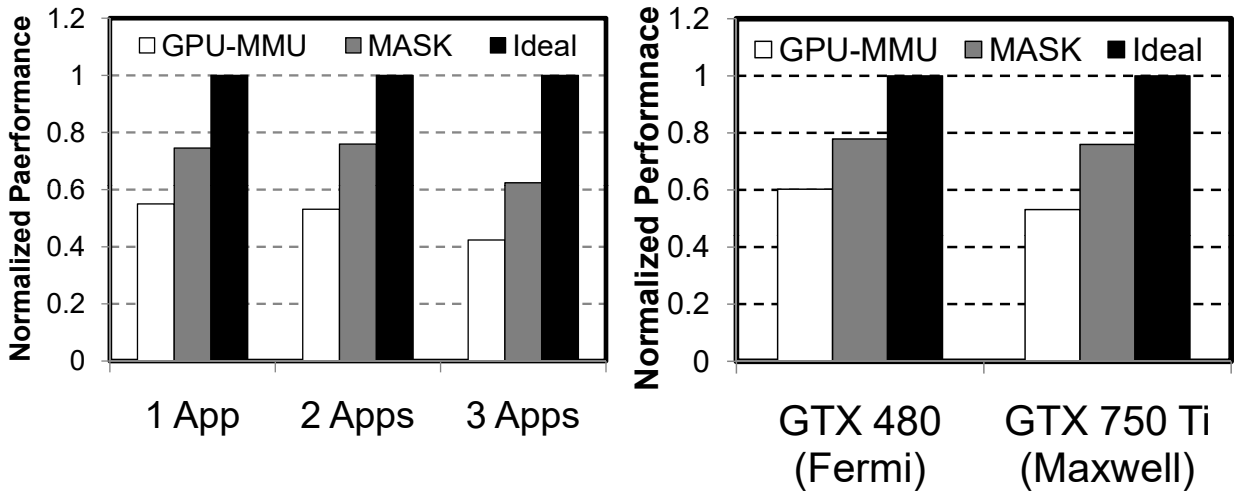
(a) DRAM Bandwidth Utilization **(b)** DRAM Latency

Figure 6.19. DRAM bandwidth utilization and latency.

6.6.3. Scalability and Performance on Other Architectures

Figure 6.20a shows the performance of GPU-MMU and MASK, normalized to the ideal performance with no translation overhead, as we vary the number of applications executing concurrently on the GPU. We observe that as the application count increases, the performance of both GPU-MMU and MASK are further from the ideal baseline, due to contention for shared resources (e.g., shared TLB, shared data cache). However, MASK provides increasingly better performance compared to GPU-MMU (35.5% for one application, 45.7% for two concurrent applications, and

47.3% for three concurrent applications). We conclude that MASK provides better scalability with application count over the state-of-the-art designs.



(a) Scalability analysis **(b)** Performance on Fermi

Figure 6.20. Scalability (a) and portability (b) studies for MASK.

The analyses and designs of MASK are architecture independent and should be applicable to any SIMD machine. To demonstrate this, we evaluate MASK on the GTX 480, which uses the Fermi architecture [271]. Figure 6.20b shows the performance of GPU-MMU and MASK, normalized to the ideal performance with no translation overhead, for the GTX 480 and the GTX 750 Ti. We make three observations. First, address translation incurs significant performance overhead in both architectures for the baseline GPU-MMU design. Second, MASK provides a 29.1% performance improvement over the GPU-MMU design in the Fermi architecture. Third, compared to the ideal performance, MASK performs only 22% worse in the Fermi architecture. On top of the data shown in Figure 6.20b, we find that MASK reduces unfairness by 26.4% and increases the TLB hit rates by 64.7% on average the in Fermi architecture. We conclude that MASK delivers significant benefits regardless of GPU architecture.

6.6.4. Hardware Overheads

To support memory protection, each L2 TLB cache line adds an address space identifier (ASID). In our evaluation, we added one byte per TLB entry for ASID bits, which translates to 7% of the L2 TLB size.

To support TLB-FILL TOKENS, we add two 16-bit counters at each shader core. In the shared cache, we add a 32-entry fully-associative content addressable memory (CAM) for the bypass cache, and 30 15-bit token counts with 30 1-bit token direction entries to support the token distribution for 30 concurrent applications. In total, we add 436 bytes of extra space (4 bytes per core on the L1 TLB, and 316 bytes in the shared L2 TLB). This additional overhead translates to 0.5% growth of the L1 TLB and 3.8% of the L2 TLB.

TLB-REQUEST-AWARE L2 BYPASS support adds ten 8-byte counters per core to track cache hits and cache accesses per level, including the data cache hit and accesses. This is 80 bytes total and is less than 0.1% of the shared L2 cache. Each cache and memory request contains an additional 3 bits specifying the page walk level, as discussed in Section 6.4.3.

ADDRESS-SPACE-AWARE DRAM SCHEDULER adds a 16-entry FIFO queue in each memory channel for TLB-related requests, and a 64-entry memory request buffer per memory channel for the SILVER QUEUE, while reducing the size of the NORMAL QUEUE by 64 entries down to 192 entries. This adds an extra 6% of storage overhead to the DRAM request queue per memory controller.

6.7. MASK: Conclusion

Efficiently deploying GPUs in a large-scale computing environment needs spatial multiplexing support. However, the existing address translation support stresses a GPU's fundamental latency hiding techniques, and interference from multiple address spaces can further harm performance. To alleviate these problems, we propose MASK, a new memory hierarchy designed for multi-address-space concurrency. MASK consists of three major components that lower inter-application

interference during address translation and improve L2 cache utilization for translation requests. MASK successfully alleviates the address translation overhead, improving performance by 45.7% over the state-of-the-art.

Chapter 7

Reducing Inter-address-space Interference with Mosaic

Graphics processing units (GPUs) are the tool of choice in an ever-growing range of application domains due to steady increases in compute density and better front-end programming tools [12, 194, 274]. While the programmability of GPUs has clearly benefited from better high-level language support [59,274,322,358], dramatic additional improvements are enabled by emerging virtualization features long taken for granted in CPUs, such as unified virtual address space support [12, 271], demand paging [276], and preemption [9, 276]. Such familiar features can dramatically improve programmer productivity, but introduce difficult tradeoffs in GPU design.

GPU memory virtualization has yet to achieve performance similar to CPUs due to a number of significant fundamental challenges [241, 375]. Memory virtualization relies on page tables to store virtual to physical address translations and uses translation lookaside buffers (TLBs) to further reduce the cost of address translation. In GPU-based systems, the highly concurrent, SIMT execution model of GPUs, which relies on multi-threading and asynchrony to hide memory latency, is at fundamental odds with the synchronous memory accesses required to translate virtual addresses on demand [73,302,303]. For discrete GPUs, page faults for data that is not resident in GPU memory can require a further synchronous PCIe data transfer [291], further harming performance [405]. More-

over, the state-of-the-art in GPU memory virtualization relies on hardware-controlled mechanisms and policies to implement memory protection and memory sharing across memory protection domains such as different processes or different virtual machines [9,241,271,272,273,276,375]. This reliance on hardware-controlled mechanisms sacrifices the ability to implement flexible policy in software enjoyed by operating-system-based memory management techniques commonly used in CPUs.

As shown in Chapter 6, the current state of the art TLB designs [73, 302, 303] suffer fundamentally from *inter-address-space* interference and limited capacity. While application working sets and DRAM capacities have exploded in recent years, TLBs have not. Consequently, TLB can only cover a small fraction of the total translations required by an application. This problem is called poor TLB reach, which leads to poor performance especially on GPU-based systems. In a GPU, a TLB miss is particularly costly, as a single TLB miss can stall dozens of threads at once, undermining the latency-hiding techniques GPUs rely on to support high computational throughput as shown in Chapter 6 and previous works [241, 375].

Large memory pages, which have been supported by CPUs for decades [336, 342] are an attractive solution. Because a large page can be several orders of magnitude larger than a small page (e.g. 4KB vs. 2MB or 1GB in modern CPUs [164, 166]), a fixed number of translations cached at page granularity cover a significantly larger fraction of a virtual address space when pages are larger. However, large pages have well-known drawbacks which have limited their adoption even in CPUs. They increase the risk of internal fragmentation, as it is often difficult for an application to completely utilize large contiguous regions of memory, leading to bloat and unpredictable memory access latencies [204]. Moreover, for demand-paged GPUs [276], larger pages mean larger PCIe latency for page-fault-driven memory transfers (see Section 7.2), and risk transferring more data than an application actually uses.

Demand-paging is a compelling feature that can actually improve performance. We minimally modified programs from the Rodinia benchmark suite to use NVIDIA's unified memory and demand paging and find that demand-paged memory actually *improves* performance for some

programs, such as the Gaussian benchmark. While demand-paged memory does not improve performance for all workloads, the programmability advantages are compelling, providing strong incentive to improve its performance in future GPU architectures. However, in stark contrast to address translation, demand paging performance is best served by smaller pages sizes, which reduce transfer latencies, and give greater flexibility to prefetchers and thread schedulers to mask or overlap those latencies. An apparent fundamental tension exists: larger page sizes can improve address translation performance but harm demand paging, and vice-versa [303, 405].

Our goal in this work is to find techniques that address this apparent fundamental tension with transparent large page support for GPUs. By transparently using multiple page sizes, and changing virtual address space mappings dynamically, the system can support good TLB reach with large pages for GPGPU applications, along with better demand paging performance based with small pages. While transparent large page support has proved challenging for CPUs [204, 265], a number of observations about the memory behavior of contemporary general-purpose GPU (GPGPU) applications (see Section 7.2) suggest that effective GPU-based implementation may side-step many of those challenges. A *key observation* is that the vast majority of memory allocations/deallocations are performed *en masse* by GPGPU applications in phases, typically soon after launch or before exit, and long-lived memory objects that increase fragmentation and induce complexity in CPU memory management are largely absent in the GPU setting. Moreover, the predictable memory access patterns of GPU codes simplify the task of detecting when a region can benefit from large page mappings. When data are copied over from the CPU memory to the GPU memory, initial physical mappings at small granularity can be coalesced after transfers complete, provided that the backing physical memory is already contiguous. By designing to ensure this is the common case, we can minimize fragmentation and simplify implementation of primitives for coalescing and splintering pages.

This dissertation presents a design and a prototype for Mosaic, a GPU memory management design that embodies these insights. Mosaic relies on predictive memory layout policies to preserve contiguity, which avoids page splintering and coalescing costs when possible, and reducing

their performance impact when they are necessary. The key design components of Mosaic are 1) **CoCoA**, which provides a memory mapping to avoid the performance overhead of promotion and demotion; 2) Lazy-Coalescer, a hardware-software cooperative page size promotion and demotion primitive; and 3) CaC, which transparently modifies mappings and moves data to avoid fragmentation. Mosaic not only tracks spatial and temporal locality to inform the promotion and demotion of large pages [204], but also proactively promotes blocks of memory when possible, which allows Mosaic to avoid fragmentation while improving TLB reach.

This dissertation makes the following contributions:

- We analyze fundamental trade-offs between TLB reach, demand paging performance, and fragmentation, motivating the need for the transparent support of multiple page sizes on GPUs.
- We present a hardware-software cooperative design for managing and using large pages on a discrete GPU. Our design includes mechanisms for splintering and coalescing base pages from/to large pages, that allows concurrent accesses to pages undergoing these operations on other GPU streaming multiprocessors (SMs).
- We show that transparent support for large pages (e.g., pages that are 2MB or larger) improves TLB reach to the degree that makes fine-grained memory protection and cross-address-space concurrency both scalable and practical for GPUs.
- We evaluate a prototype implementation of our design, Mosaic, and show that our multi-page-size design provides 46.7% performance improvement over a state-of-the-art GPU memory manager.

7.1. Background

Hardware-supported memory virtualization relies on address translation to map virtual addresses to physical addresses and uses TLBs to cache these mappings and reduce the effective

latency for accessing page tables. While technological trends have driven a recent explosion in DRAM capacities, TLB capacities have not kept pace, and the ratio of TLB reach (the maximum amount of memory addressable through cached TLB entries) to memory capacity has been steadily declining for decades. A direct consequence is that overheads for address translation have grown to dominate run time for many important workloads with large memory footprint [46, 113, 241, 302, 303, 375]. TLB reach can be increased with support for multiple page sizes. However, providing such support in GPUs comes with several challenges, including implementing the required address translation hardware, implementing memory management without blocking synchronization primitives or background threads, and specifying and enforcing sound policies.

7.1.1. Architectural Challenges

While memory hierarchy designs for widely-used GPU architectures from NVIDIA, AMD, and Intel are not publicly available, it is widely accepted that GPUs support TLB-based address translation, multiple page sizes, and, in some models, large page sizes (2MB or larger) [9, 241, 271, 272, 273]. Recent research on TLB design for integrated GPUs [302, 303] and on GPU demand paging support [12, 15, 276, 405] corroborate our own findings on the performance cost of address translation and the performance opportunity of large pages (see Section 7.2). The integrated setting requires additional OS-level support [302, 303] because mappings must be shared between the CPU and the GPU. In the discrete setting, the benefits of 2MB pages are outweighed by their negative performance impact on demand paging [405].

7.1.2. Policy Challenges

While conceptually simple, the support for multiple page sizes introduces complexity for memory management that has been difficult to manage and, despite a decades-long history of architectural support, adoption has been slow. The availability of larger pages can either be managed transparently below a memory management interface or be exposed to application programmers. Application-exposed management forces programmers to reason about physical memory and use

specialized APIs [31,248] for page management that sacrifice portability. Transparent support (e.g. by the OS), in contrast, requires no change to existing programs to use large pages but does require an OS to make predictive decisions about whether applications will benefit from large pages or not. OS-level large page management remains an active research area [204, 265], and abundant anecdotal evidence suggests it is still a hard problem: optimization guidance for many modern applications advises strongly against using large pages [76, 251, 267, 297, 316, 347, 362, 380], due to memory bloat and unpredictable page fault latencies.

Transparent large page management encapsulates policy about whether to use large or small pages in the memory manager. The memory manager can change mappings dynamically by coalescing or splintering pages and can create contiguity by compacting pages. Due to its flexibility and lack of program-level impact, transparency is a key goal of Mosaic.

7.1.3. Implementation Challenges

Transparent large page management in the GPU inherits many of the same challenges described above. Like its CPU counterpart, transparent support on GPUs requires primitives to implement transitions between different page sizes (coalescing and splintering) and mechanisms such as compaction and reservations to create and/or preserve contiguity so that large pages are available when they are needed, along with sound policy that maximizes performance, avoids fragmentation, and minimizes overheads [265]. However, GPUs face additional challenges as they rely on hardware-based memory allocation mechanisms and management. While CPU page table management can use locks and preemption to implement atomic updates to page tables, can fault synchronously on virtual addresses whose mappings are changing, and can use background threads to perform promotion and compaction, GPUs currently offer no mechanisms to atomically move pages or to change page mappings for promotion or compaction, and these features are non-trivial to implement.

7.2. A Case for Multiple Page Sizes

This section analyses the advantages and disadvantages of using base and large pages in GPU applications, showing how transparent support for multiple page sizes can maximize the benefits of both page sizes while minimizing performance costs.

7.2.1. Baseline Design

As we observed in the previous section, the introduction of address translation hardware such as TLBs into the massively parallel GPU memory hierarchy puts TLB misses on the critical path that can degrade performance significantly. The opportunity to reduce this impact using the improved reach of larger pages to reduce TLB miss rates has been observed in other recent works on GPU TLB design [302, 303, 405]. To quantify the performance tradeoffs between using base pages and large pages, we simulate a number of designs from the recent literature that extend the TLB design proposed by Power et al. [303] to support demand paging [405], and compare them against the work by Cong et al. [73]. We modify the MAFIA framework [174], which extends GPGPUsim [41] to support cross-process concurrency. We also modify the CUDA simulator in GPGPU-Sim to implement page-based memory allocation and manage device-side page tables (see Section 7.4 for more details).

Mosaic assume similar baseline design as MASK, as shown in Figure 7.1. In this baseline design, each core has a private L1 cache and all cores share a highly-threaded page table walker (❶). On a TLB miss, the shared page table walker probes the shared cache (❷). L2 TLB lines are extended with address space identifiers and a parallel page table walker (❸). TLB accesses from multiple threads to the same page are coalesced. On a private L1 TLB miss, the shared L2 TLB is probed. On a shared L2 TLB miss, a page table walker begins a walk (❷). On a shared TLB miss, the page table walkers (❸) performs a page walk, retrieving data from the main memory (❹).

To model a system which extends this design to support demand paging, we further extend the simulation by detecting page faults taken for pages that are not yet GPU resident and by modeling the latency for the subsequent DMA transfer that would be required to fault the data in. Follow-

ing the nomenclature from [405], we denote GPU-side page faults which induce demand DMA transfers across the PCIe bus "far-faults". Our experience with GTX 1080 cards suggests that production implementations perform significant prefetching to reduce latencies when reference patterns are predictable, which is not modeled by our simulation.

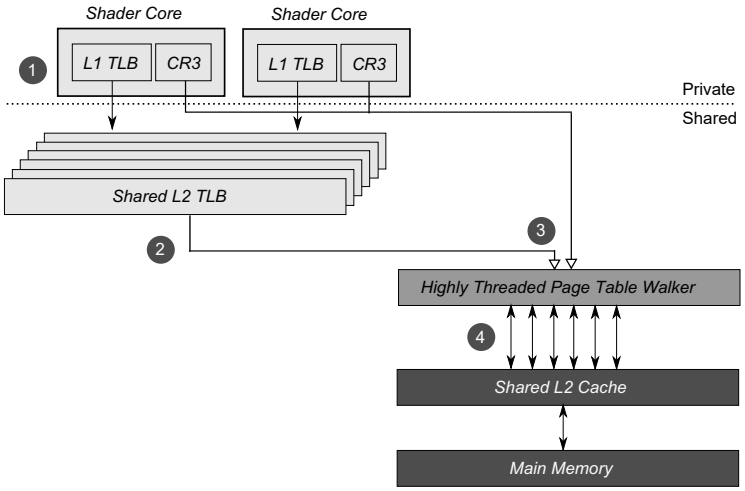


Figure 7.1. Baseline multi-level TLB design. Mosaic assumes a similar baseline as MASK

Figure 7.2 shows the performance of two baseline designs, normalized to an ideal baseline where all TLB requests hit in the L1 TLB. The data shows that optimization of the memory subsystem can minimize the overhead lost to address translation through interference. However, cases where TLB misses stall all warps remain common, incurring performance overheads of up to 48.1%.

When using 2MB pages in scenarios where no far-faults occur (hence, no data transfer overhead), using the same number of TLB entries as the 4KB design to map 2MB pages substantially reduces the TLB miss rate. With 2MB pages, the much larger TLB reach allows applications to reach within 2% of their ideal (infinite TLB) performance. Thus, the incentive to use large pages is compelling.

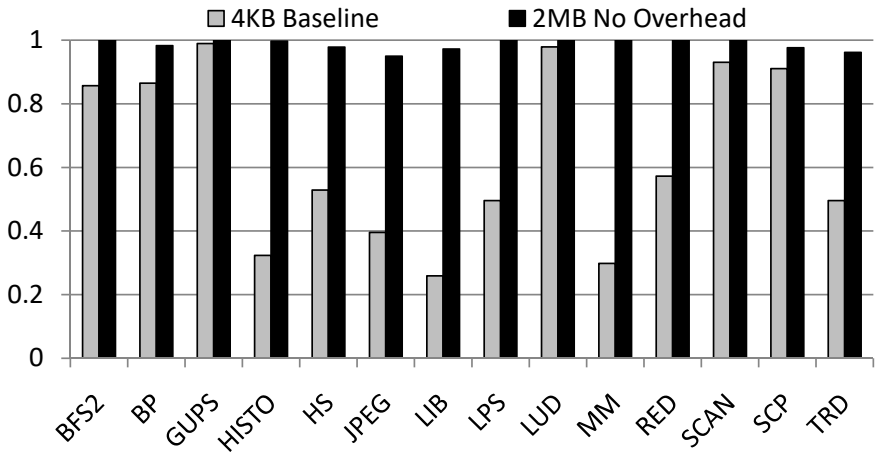


Figure 7.2. Comparison of performance between two memory protection designs: a multi-level TLB with 4KB pages and an optimized multi-level TLB with 2MB pages.

7.2.2. Large Pages Alone Are Not the Answer

A natural solution to consider is simply using *only* large pages. Attractive for its simplicity, this approach would reduce address translation overheads dramatically with minimal effort from the hardware or runtime. Unfortunately, this solution is untenable because it increases the size of all demand-paging requests, harming performance unacceptably as well as wastes memory through bloat from internal fragmentation. .

Demand Paging at 2MB Granularity. Previous work that provides support for demand paging on GPUs observes that paging at 2MB granularity lowers the number of far-faults in GPU kernel codes that exhibit locality, but has the side effect of increasing the load-to-use interval dramatically for those far-faults [405]. Using 2MB page size can also cause internal fragmentation and poor locality, further increasing the transfer overheads for data that are never used. Most importantly, for workloads with high locality, larger pages limit the ability to overlap transfer with execution, as *all* warps touching a 2MB extent must stall. Our simulations confirm these effects using load-to-use latency ratios for 4KB and 2MB pages of 6:1 (based on PCIe transfer latency measured on the GTX 1080). Across all our workloads, the overall impact is an unacceptable 93% slowdown.

Smaller page sizes also give the runtime more flexibility to implement strategies such as predictive prefetching and transfer overlapping to hide transfer latencies. To characterize the performance implications, we experiment with demand-paging microbenchmarks on a NVIDIA GTX 1080.

A series of workloads measure sequential-access and random-access load-to-use intervals over a 64MB dataset allocated as a continuous 64MB region, allocated in 2MB chunks, and allocated in 4KB pages. Figure 7.3 shows the average and maximum load-to-use intervals in microseconds on a log scale. Error bars are also included to indicate one standard deviation of the variance in the average latency.

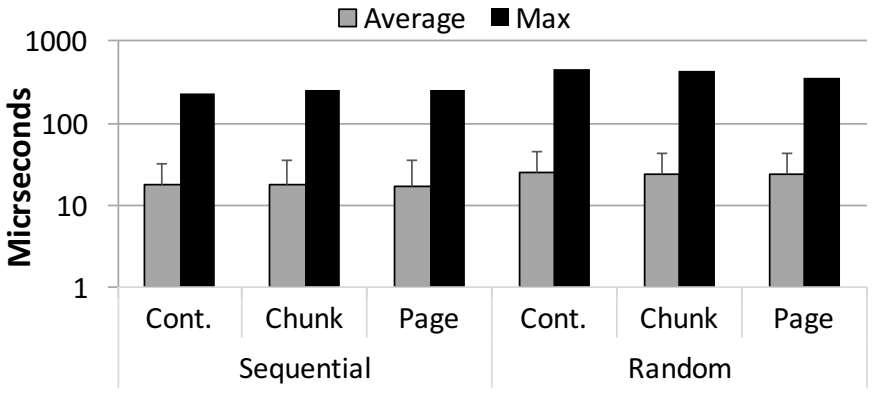


Figure 7.3. Demand paging load-to-use latencies measured on NVIDIA GTX 1080 for sequential and random access patterns using contiguous, chunked, page-granularity allocation. Y axis is in log scale. Error bars represent one standard deviation of variance.

We observe that the paging implementation is able to hide latencies and amortize transfer costs even under the most adversarial access pattern, bringing average latencies to under 18 and 25 microseconds for sequential and random accesses respectively. We also observe that allocation granularity has little effect on paging performance. However, larger pages would limit the ability of the runtime to keep effective latencies this low.

Bloat. While using only large pages could eliminate external fragmentation, it exposes the system to internal fragmentation and memory bloat. To understand the impact of bloat, we simulate all our workloads at 2MB page size in a single application setup (Section 7.4) and found that the working set size inflates *by 40.2% on average and up to 367% in the worst case*. We consider these estimates to be conservative as fragmentation would likely continue to worsen if we could simulate for longer time scales. Such waste is unacceptable, particularly when there is significant memory pressure.

We conclude that, despite the performance opportunity of large pages (2MB), 4KB base pages

can be critical to performance, particularly on demand-paged GPUs. A design that transparently delivers the best of both page sizes is needed.

7.3. Mosaic

This section describes the design of Mosaic. We first enumerate design constraints and the basic primitives required, such as coalescing, splintering, and compacting pages, and subsequently integrate them to form a complete design.

7.3.1. Design Constraints

Primitives. Modulating page sizes backing an application's virtual address space requires mechanisms for coalescing and splintering. *Coalescing* involves collecting a set of 4KB base pages into a single contiguous 2MB page in physical memory. This process can involve 1) moving memory between base pages and large pages and 2) moving and compacting data to create the required contiguity in the physical address space. *Splintering*, the reverse operation, breaks a single 2MB page into a series of 512 contiguous 4KB base pages. While splintering need not involve copy, it is seldom performed without being paired with a subsequent operation that performs a copy (e.g., compaction, copy-on-write, or data transfer across PCIe, etc.). Both operations require updates to page table mappings which must appear atomic to the application. In CPU-based large page management, these operations are performed by the operating system, which can either commandeer an application's execution context or use background threads to do it. The atomicity of these operations is achieved using a combination of locking and inter-processor interrupts (IPIs).

On top of this, our design must contend with limited support for synchronization along with the GPU's high levels of concurrency. In particular, memory accesses to virtual pages whose mappings are being updated must *stall* until the update completes, and stale TLB entries must be flushed. As GPUs are incapable of leveraging background threads to perform operations such as compaction, it argues for a design that proactively minimizes the need to be handled through background threads.

Concurrency. Ability to support cross-address-space protection is a major motivator for im-

plementing address translation, a design that handles concurrent address translation well is critical. Depending on allocation policy, 2MB extents of physical memory may contain 4KB pages from different address spaces (from different GPGPU applications). This can force memory copy to create contiguity under fragmented conditions. Alternatively, a design that never shares 4KB pages in a 2MB physical extent across protection domains (i.e., all 4KB pages within the 2MB range have the same address space identifier) can sidestep the need for copying at the cost of increased susceptibility to fragmentation. Mosaic's design reflects an imperative to allocate physical memory across applications to proactively conserve contiguity, avoid copying during coalescing and splintering. While previously-proposed DRAM techniques that accelerate copy with bulk primitives [64, 331] are available, copying data across DRAM banks incurs a significant performance impact and should be avoided.

Correctness. Atomicity of updates to memory mapping or changes to the physical layout backing a virtual address space must be preserved from the perspective of the application. Mosaic's design prohibits memory accesses by application threads that may use a stale or partially updated address translation through a combination of TLB shootdowns and blocking all translation requests at the TLB. Both techniques are costly for performance, so strategies for minimizing their usage and overhead per operation are necessary.

7.3.2. Design of Mosaic

Mosaic consists of three components: **CO**NTIGUITY-**CO**NSERVING **A**LLOCATION (**COCOA**), LAZY-COALESCER, and **C**ONTIGUITY-**A**WARE **C**OMPACTION (CAC), which work in concert to provide transparent large page support with minimal common case overhead.

The memory allocator plays an important role in conserving contiguity as well as eliminating data copies for coalescing. When coalescing a page, data copy is necessary only when the underlying contiguity in the physical address space is not already present (e.g. because pages must be copied into the physical region and any other currently active pages must be moved out of the way). Figure 7.4 shows an example in which there are three pages under the physical large

page range that are mapped from Process 2 while the remaining pages in the large page range are backing pages in Process 1. To create contiguity to enable a large page mapping, Process 2's pages need to be *moved, remapped, and replaced* by pages from Process 1, causing six page-size data movements as well as remapping overhead. To avoid such overheads, Mosaic's memory allocator, called **CONTIGUITY-CONSERVING ALLOCATION** (**COCOA**), maintains an important soft-guarantee: memory allocation from different processes can never fall within the same large page range.

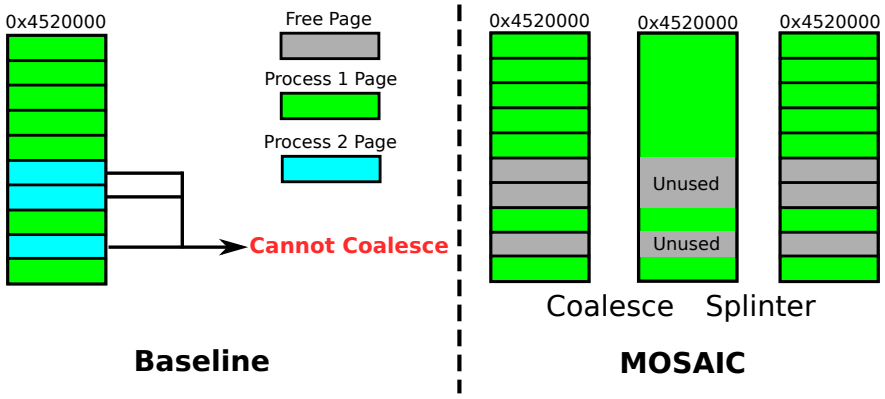


Figure 7.4. Memory allocation in Mosaic aims to avoid sharing large page ranges between multiple processes, reducing the overhead of creating large pages.

As shown in Figure 7.4, **COCOA**forces the large page range to only contain base pages from Process 1. The clear advantage of maintaining this guarantee is that, in the common case, applications do not share page frames within a large page range, allowing coalesce operations to proceed without the need for memory copies.

COCOAhas the disadvantage of risking application-level fragmentation by reserving multiple large pages while they are only partially-allocated. Our simulations (Rodinia) do not run long enough to give rise to the levels of fragmentation that would force the allocator to preempt some of these reservations to release physical memory, but we can simulate these conditions by fragmenting the memory before running the simulations. Compaction is performed by **CONTIGUITY-AWARE COMPACTION** (CAC) when fragmentation reaches a predetermined threshold. (see Section 7.3.2.1). Section 7.5.3 provides a thorough analysis of these effects.

Coalescing and Splintering. As we mentioned, Mosaic maintains a soft guarantee where, in

the common case , the allocator will prevent processes from sharing a large page range. Mosaic's contiguity-aware memory allocator enables common-case coalescing and splintering to require *no* memory copy operations. In this case, the main tasks that the memory manager has to do are 1) ensure all leaf pages within the large page are in the proper order and 2) ensure that all the leaf pages are disabled so nothing can access an invalid small page. For example, suppose that the page table walking process has four levels of page table entries. To create a large page mapping, the memory manager atomically swaps the 8-byte L3 page table entry, currently pointing to an L4 page table page containing leaf PTEs for all the base pages in the corresponding large page range, with a single 2MB leaf PTE. The memory manager marks the L4 page table page as "disabled" but retains its entries in order to enable a similar fast path for splintering. Stale TLB-resident translations for base pages in the new large page still provide the same virtual-to-physical translation, so TLB shootdowns can be avoided as an optimization. Hence, the only overhead during this process is the DRAM writes to update the L3 page table entries. Splintering performs the inverse operation. As a result, with Mosaic, coalescing and splintering do not trigger TLB flushes or invalidations. Additionally, the latency of DRAM writes from page table modifications can be overlapped with the page table walk from the first large page compulsory miss, leading to a significantly smaller performance overhead, as shown in Figure 7.5.

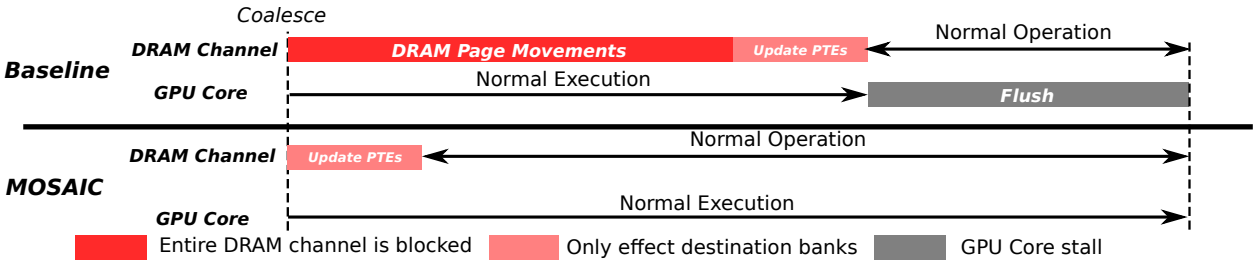


Figure 7.5. The benefit of LAZY-COALESCER compares to the state-of-the-art baseline. In the baseline case, virtual-to-physical mapping has to be modified, multiple pages within DRAM has to be moved and GPU cores need to be flushed before the MMU can form a large page.

In the case where coalescing requires a TLB shootdown because there are multiple ongoing page walks to the same large page range, coalescing triggers the TLB shootdown. In this case, the TLB MSHR entry responsible for the ongoing page walk triggers the memory management unit, which then sends flush signals to all the GPU cores to stall all memory requests.

Updating the page table entries allows the hardware memory management unit to keep track of what pages are coalesced. As a result, this enables transparent support as the software side (GPGPU applications) does not have to be aware of the large pages. However, this comes at a cost, as the hardware side (i.e., each GPU SM) has to be able to probe into the correct TLB before it is known whether an address falls under a large or small page. To circumvent this problem, we 1) invalidate the large page TLB whenever there is a splintering command so that accesses to small pages never hit in the large page TLB and 2) probe the large TLB first before proceeding to the small TLB.

Transparency Support. Because the software-side is responsible only for performing memory allocation, both the CUDA and OpenCL runtimes can support **CoCoA**without changes to the GPGPU applications that are running. In the rare case that the GPU hardware is unable to perform coalescing, the GPU can still translate the same virtual address (i.e., there is no change to the virtual-to-physical address mapping of the base page). Lastly, because the soft guarantee is set, coalescing never triggers any data movement.

Determining When to Coalesce Pages. Coalescing and splintering decisions are typically based on the hotness of base pages within a large page. As observed in previous literature [73], threads across multiple warps maintain good spatial locality under multiple adjacent small pages because these threads typically operate on the same large array or set of data that are allocated together. We observe similar behavior in our evaluation. Hence, we simplify the hardware by determining when to coalesce a page proactively during the allocation phase by using how many base pages are being mapped within the large page range.

Proactively Coalescing Pages. Most of the time, a program can allocate a new region of memory of any size at any time during its execution. However, in GPGPU applications, we observe that in order to limit the number of data transfers over PCIe, GPGPU programs groups GPU memory allocations together. Typically, GPGPU applications are broken down into phases: data alloca-

tion, data preparation, kernel execution, and data retrieval. As a result, data allocations, which are managed by the host memory allocator, are typically known before the kernel is launched to the GPU. Hence, it is possible to modify the GPU memory allocator to provide software support for multi-page-size TLB design.

While the MMU can extract the page utilization information by probing the access bit in the page table entries,[1] Mosaic goes a step further. In this work, we utilize our observation on the memory allocation behavior of GPGPU applications. Specifically, we observed empirically that the base page mapping information in combination with an LRU policy can infer the utilization of a page. This means that many coalescing decisions can be determined at allocation time.[2] When the kernel containing these coalesced pages is dispatched and the data is copied from the CPU-side memory to the GPU-side memory, the MMU then performs coalescing right away. This provides an additional benefit, as preemptively coalescing pages reduces the amount of thrashing on the small page TLBs.

Determining When to Splinter a Large Page. Because the coalescing decision is made proactively using the base page mapping information, we can further simplify the hardware by deferring splintering until the page is deallocated. If deallocation frees up only a part of a large page, LAZY-COALESCER will splinter the remaining pages back into small pages.

Demand Paging. Lastly, to provide support for demand paging, we restrict the size of the page so that only small pages are allowed to be demand paged. To enforce this restriction, if a large page is touched by a demand paging operation, we first splinter the page so that demand paging is only performed on a subset of the needed small pages.

7.3.2.1. Dealing with Data Fragmentation

As shown in Figure 7.4, the combination of **COCOA** and LAZY-COALESCER can increase data fragmentation by coalescing unallocated pages under a large page. We introduce **C**ONTIGUITY-

[1] This can be done in recent GPU architecture through the Falcon coprocessor.

[2] Note that only the coalescing decision is determined during allocation and that, while the MMU is aware of these coalesced commands, the MMU still behaves as if these pages are small pages until the coalescing command is committed.

AWARE COMPACTION (CAC) to Mosaic to address any rare cases where memory compaction is needed to reduce data fragmentation.

Reducing Fragmentation Through Memory Compaction. Traditionally, memory compaction is done by rearranging base pages to form a tight memory region where there is no unallocated memory between base pages. Figure 7.6 shows an example of the compaction process. For the baseline in our example, multiple pages under three different large page ranges (A, B and C) go through a compaction process. In the baseline, pages under the large page range C are moved to large page ranges A and B. The resulting page arrangement prevents coalescing in large page ranges A and B without requiring additional data movements.[3]

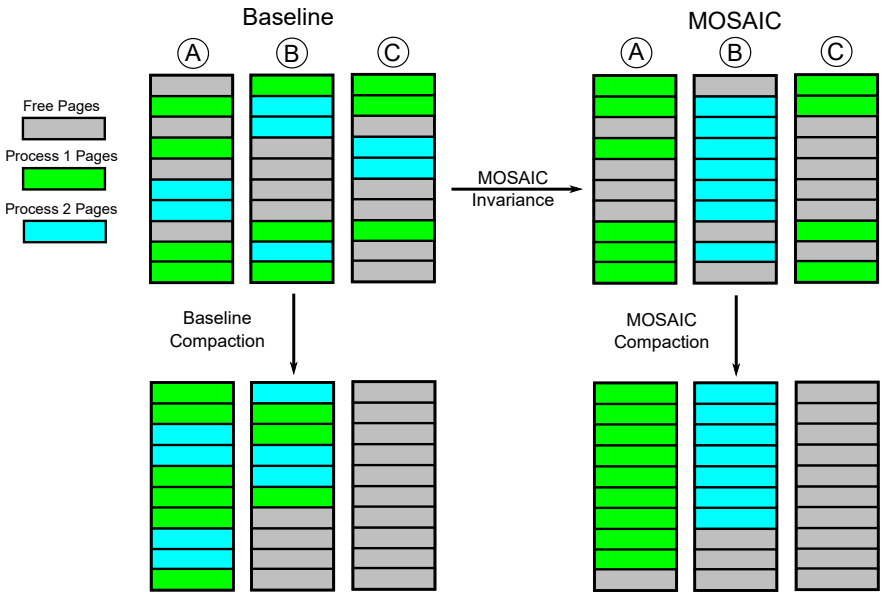


Figure 7.6. The difference between the baseline compaction mechanism and CAC showing that CAC always preserve soft-guarantee imposed by **COCOA**.

In Mosaic, CAC is modified to maintain our guarantee that large pages only contain pages from the same process by sorting base pages into new large pages based on their process ID. This ensures all pages in each large page range (A and B) belong to a single application, as shown in Figure 7.6. Because CAC forces compaction to happen locally (i.e., pages from different applications cannot mix within the same large page range), CAC does not completely eliminate all fragmentation as

[3]In the case of contiguous virtual addresses that span multiple pages, there must be enough contiguous virtual space for compaction to proceed in both the baseline design and Mosaic.

shown in 7.6. However, in the worst case, this fragmentation only happens to the last large page of each application.

Bulk Copy. To further optimize the performance of CAC, it is possible to utilize two in-DRAM bulk copy techniques [64, 331] to reduce the latency of page-level copy. This can be done by limiting the compaction process to a single memory channel. As a result, a compaction that happens in one memory channel will not stall memory requests (or shared cache requests) to other memory partitions (and channels).

7.3.3. Overall Design of Mosaic

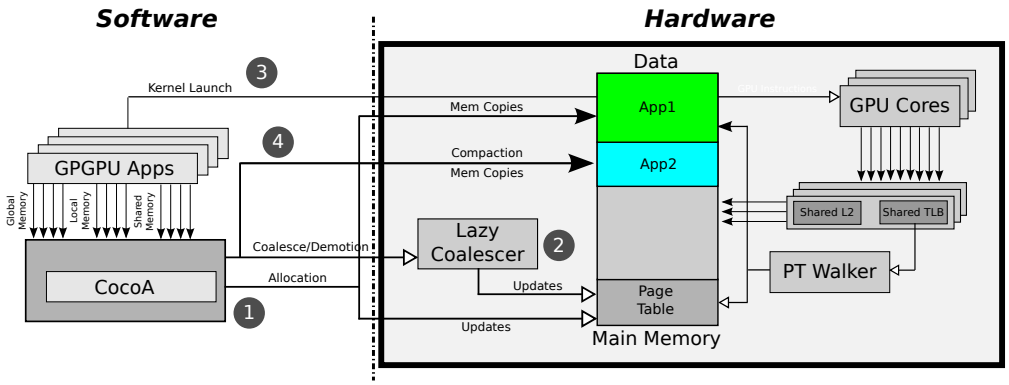


Figure 7.7. The overall design of Mosaic. (1) **CONTIGUITY-CONSERVING ALLOCATION** provides the soft-guarantee allowing LAZY-COALESCER to coalesce pages transparently. (2) coalesce requests are sent to LAZY-COALESCER prior to kernel launch (3). (4) **CONTIGUITY-AWARE COMPACTION** provides physical space contiguity and reduces the amount of fragmentation.

Figure 7.7 shows the overall design of Mosaic. Upon reaching a memory allocation call from a GPGPU application, **COCOA**performs memory allocation while preserving the soft guarantee that a large page range can only contain pages from the same application (❶). Then, if the software runtime observes an opportunity to coalesce a page, it sends a coalescing command to LAZY-COALESCER in order to perform low-overhead memory coalescing (❷). This allocation process continues until the GPGPU application reaches its kernel launch (❸). Mosaic maintains these execution flows until an application wants to deallocate memory. When there is a deallocation, Mosaic performs page splintering by sending a splinter command to LAZY-COALESCER (❷). Afterward, the memory allocator checks for memory fragmentation. If the fragmentation exceeds

a certain threshold, Mosaic performs memory compaction, as described in Section 7.3.2.1 (❹).

7.4. Methodology

We utilize a modified version of the MAFIA framework [174], based on GPGPU-Sim 3.2.2 [41], to evaluate our proposed memory manager. We modify the cuda-sim [41] memory allocator to model both **CoCoA**and the Unified Address Space [271]. We modified the MAFIA framework to model the performance impact of a page table walker and the TLB structures. Table 7.1 provides our GPU configuration details.

Shader Core Config	30 cores 1020 MHz, GTO warp scheduler [318]
Private L1 Cache	16KB, 4-way associative, LRU, L1 misses are coalesced before accessing L2, 1 cycle latency
Shared L2 Cache	2MB total, 16-way associative, LRU, 2 cache banks 2 interconnect ports per memory partition, 10 cycle latency
Private L1 TLB	128/32 (base page/large page) entries per core, fully associative, LRU, single port 1 cycle latency
Shared L2 TLB	512/8 (base page/large page) entries, 16-way/fully-associative (base page/large page), LRU 2 ports per memory partition, 10 cycle latency
DRAM	GDDR5 1674 MHz, 6 channels, 8 banks per rank FR-FCFS scheduler [317, 406] burst length 8
Bulk Copy	1366 ns. baseline, 83.75 ns. inter-SA latency [64, 331] 260.5 ns. inter-bank latency [64]
Page Table Walker	64 threads shared page table walker, traversing 4-level tree structure similar to previous work [303]

Table 7.1. The configuration of the simulated system.

Address Space Within the Unified Virtual Address. We modify GPGPU-Sim [41] to model the behavior of the Unified Virtual Address Space [271]. We add the memory allocator into cuda-sim to handle all virtual-to-physical address translation. Our simulator ensures memory protections not only across different GPGPU applications, but also across all type of memory references (global, local, and shared).

TLB and Page Table Walk Model. We modify the MAFIA framework to accurately model the Mosaic baseline design. Each core has a private L1 TLB. The page table walker is shared and admits up to 64 threads of concurrent walks. The baseline design for Mosaic adds a shared L2 TLB instead of page walk caches, with a shared L2 TLB located in each memory partition. Both L1 and L2 TLB entries contain MSHR entries to track in-flight page table walks. On a TLB miss, a page table walker generates a series of dependent requests that probe the L2 data cache and main memory as needed. To correctly model the physical addresses for dependent memory accesses, we collect traces of all possible virtual addresses for each application executed to completion, and pre-populate a partition of the physical address space with the contents of a page table that map the virtual address space exclusively with 4KB pages, following the layout strategy of the memory allocator described in Section 7.3.3.

Modeling Compaction. We model the performance overhead of compaction conservatively by stalling the entire GPU then flushing the pipeline. Our performance model takes into account of all data movements and writes to page tables. This gives the worst-case performance impact of the compaction mechanism proposed in this work.

Fragmentation Analysis. To provides further analyses on how Mosaic handles data fragmentation, we allow the memory allocator to pre-fragment a fraction of the main memory. The pre-fragmented data are scattered throughout the physical memory. These data do not conform to Mosaic's soft-guarantee and they can stay in other applications' large page range. We define fragmentation index as a percentage of large pages that are going to have pre-fragmented data. This pre-fragmented data must be moved out in order for the LAZY-COALESCER to coalesce into a large page.

Workloads. We form our homogeneous workloads using multiple copies of the same application. We build 27 homogeneous workloads using GPGPU applications from the Parboil [351], SHOC [80], LULESH [185,186], Rodinia [68], and CUDA SDK [270] suites. We form our heterogeneous workloads by selecting a number of random applications out of these 27 GPGPU applications. In total, we evaluate 100 different heterogeneous workloads. Out of these 100 workloads, we

further break them down to workloads with 2, 3, 4, or 5 concurrently-running GPGPU applications.

Evaluation Metrics. We report performance using weighted speedup [97,98] using the following equation: $\sum \frac{IPC_{Shared}}{IPC_{Alone}}$ where IPC_{alone} is the IPC of an application that runs on the same number of shader cores using the baseline state-of-the-art configuration [303], but does not share GPU resources with any other applications, and IPC_{shared} is the IPC of an application when running concurrently with other applications.

Fragmentation is reported as the percentage of the physical address space that is mapped by the virtual address space, but unused by applications and therefore unavailable to the memory manager to satisfy allocation requests.

Scheduling and Partitioning of Cores. As scheduling is not the focus of this research, we minimize its impact on our evaluation, and assume the equal partitioning of GPU cores across applications.

7.5. Evaluation

In this section, we provide the evaluation of Mosaic compared to two designs: the state-of-the-art baseline design previously proposed in [303] and the ideal case where every address translation results in L1 TLB Hit.

7.5.1. Homogeneous Workloads

Figure 7.8 shows the performance of Mosaic in homogeneous workloads (see Section 7.4 for more detail). We observe that Mosaic is able to recover most of the performance lost from the overhead of address translation in homogeneous workloads. On average, Mosaic is able to recover the performance to within 5.7% of the ideal performance (where there is no address translation overhead), and is able to provide a substantial 46.7% performance improvement over the state-of-the-art baseline.

We observe that Mosaic provides good scalability, and performs very close (between 33.4% to 0.5% slowdown) to the ideal baseline for 2–4 concurrent applications. However, as the number

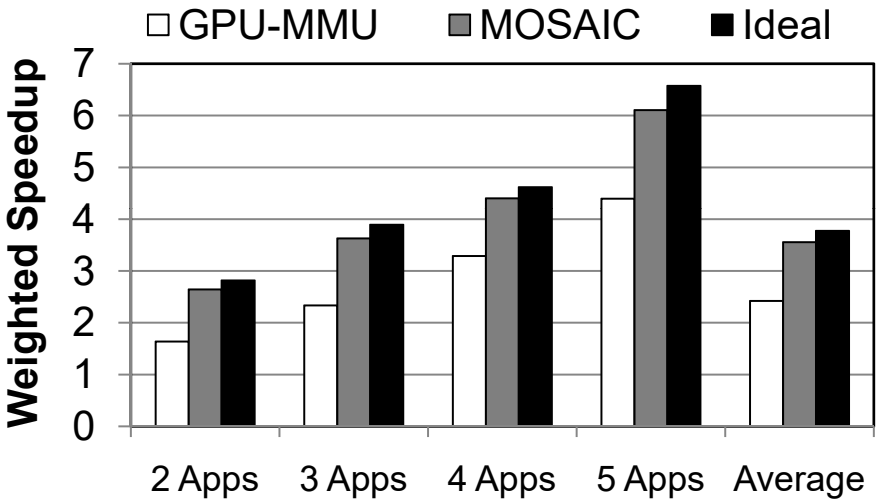


Figure 7.8. Homogeneous workload performance with a different number of concurrent applications.

of concurrent application increases, we observe that the contention at the shared TLB causes the additional performance gap between Mosaic and ideal to 7.1% on average when five GPGPU applications are running concurrently.

7.5.2. Heterogeneous Workloads

Figure 7.9 shows the performance of Mosaic in workloads consisting of randomly-selected GPGPU applications. On average, Mosaic provides 30.1% performance improvement over the state-of-the-art baseline, and performs within 14.5% of the ideal performance. We observe that the improvement comes from the significant improvement in TLB miss reduction, as shown in Figure 7.11.

To further analyze each individual workload's performance, Figure 7.10 provides 15 random samples from the 2 application setup from Figure 7.9. As shown in Figure 7.10, most applications benefit from utilizing the large page as TLB hit rate significantly increases (as shown in Figure 7.11). However, we observe that in many cases, there is still a performance gap between Mosaic and the ideal design even though Mosaic significantly improves the TLB hit rate (as shown in Figure 7.11. We discover two main factors that lead to this performance gap. First, we found that for workloads that are sensitive to shared L2 data cache misses (such as HS-CONS and NW-

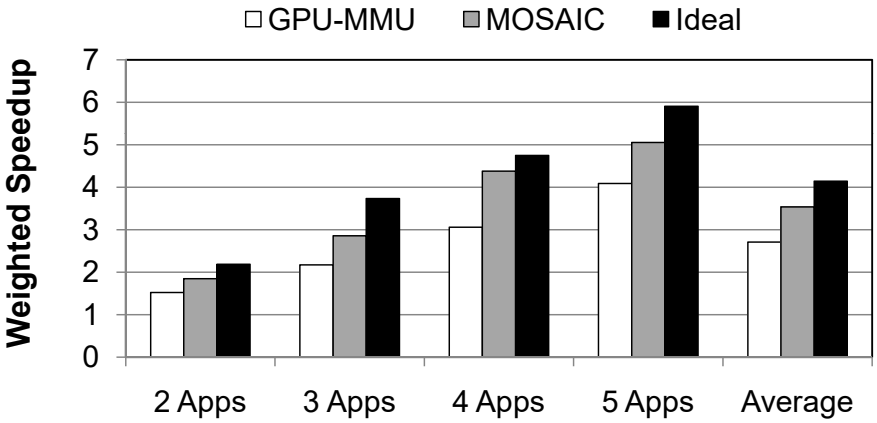


Figure 7.9. Heterogeneous workload performance with a different number of concurrent applications.

HISTO), caching data from the page table walk (due to a TLB miss) causes extra interference in the shared L2 data cache. This reduction in the shared data cache leads to the performance difference between Mosaic and the ideal design because there is no page table walk in the ideal design. Second, compulsory misses as well as the shared TLB access latency attribute to the extra performance overhead compared to the ideal design.

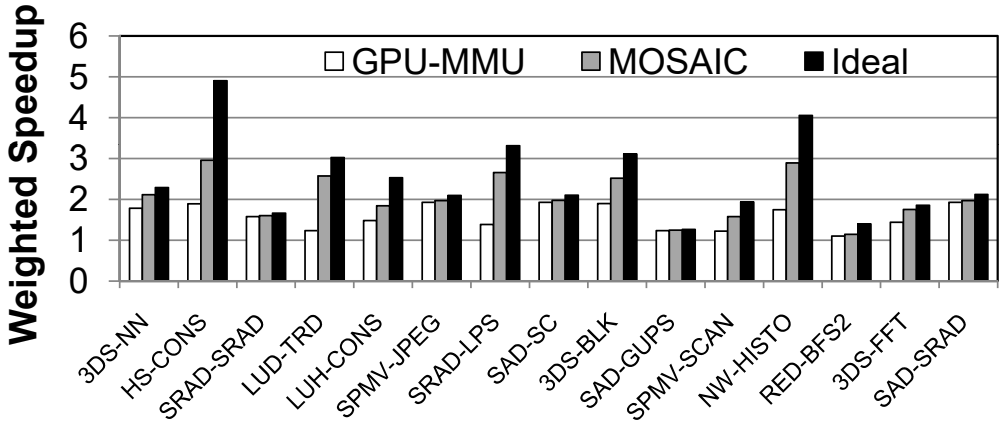


Figure 7.10. The performance of randomly selected workloads.

The impact of Mosaic on TLBs. Figure 7.11 compares the overall TLB performance of the baseline state-of-the-art with 4KB pages [303] to Mosaic, on workloads that have a shared TLB hit rate lower than 98% (i.e., workloads that suffer from limited TLB reach). From Figure 7.11, we provide two observations. First, Mosaic is very effective in increasing the TLB reach of these workloads, resulting in a massive TLB miss reduction down to 1% on average in both the L1 and

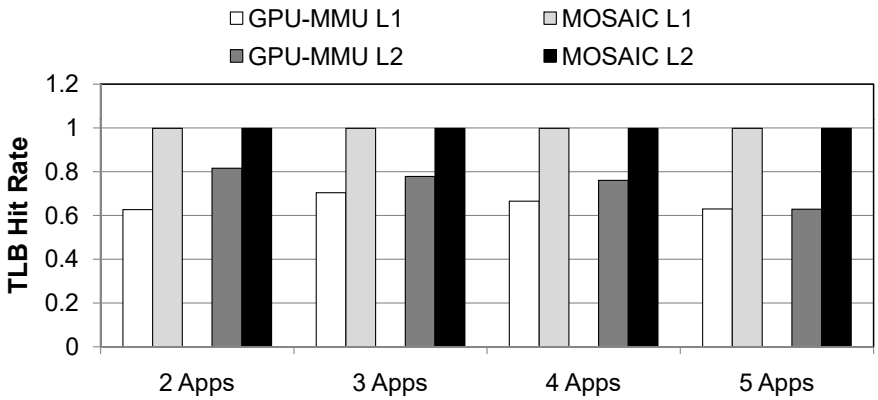
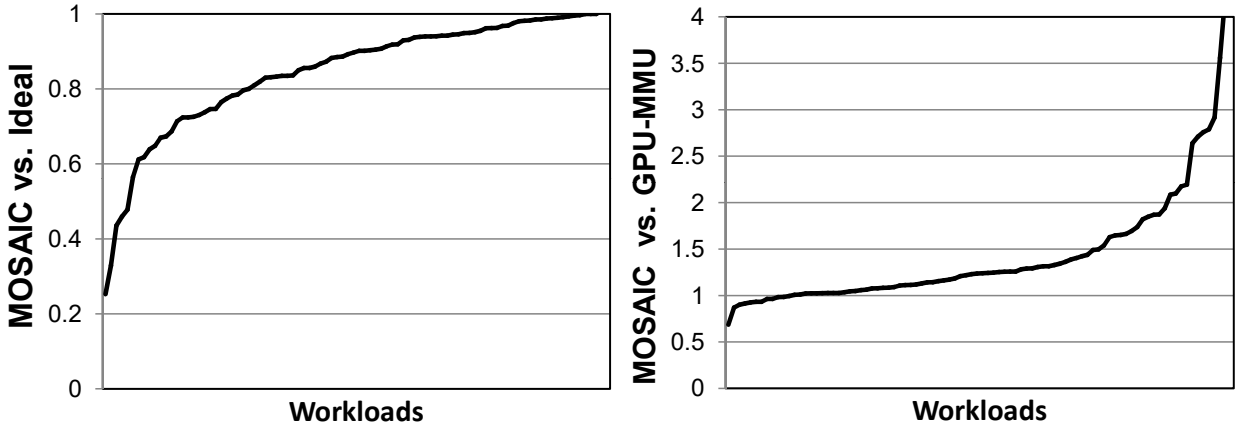


Figure 7.11. TLB characteristics of Mosaic.

shared TLBs. Second, we observe an increasing amount of interference in the baseline design when more applications are running concurrently. This results in a lower TLB hit rate as the number of applications increases. The shared TLB hit rate drops from 81% in workloads with two applications to 62% in workloads with five applications.

While Figure 7.10 shows several samples of individual performance comparison between different designs, Figures 7.12a and 7.12b provide the performance comparison for all individual workload we evaluated from all configurations (235 workloads in total). From Figure 7.12a, we observe that Mosaic is able to limit the performance impact of address translation to less than 10% for more than half of the workloads we evaluated. Second, we observe that in cases where both the GPU-MMU baseline and Mosaic's performance falls far short of the ideal performance, these workloads has a significant amount of compulsory misses (up to 5.7% of the total TLB accesses). These two observations suggest that while Mosaic is effective in increasing the range of TLBs, reducing the latency of page table walk is an important next step to improve the performance of these GPGPU applications. Third, we observe from Figure 7.12b that in most cases, Mosaic outperforms the state-of-the-art GPU-MMU baseline by up to a factor of 6 (clipped from the plot), mainly due to the improvement in the overall TLB miss reduction as shown in Figure 7.11.



(a) Mosaic vs ideal. **(b)** Mosaic vs. GPU-MMU.

Figure 7.12. The performance of Mosaic across 235 workloads. (a) shows the performance of Mosaic relative to the ideal performance with no address translation overhead. (b) shows the performance of Mosaic relative to previously proposed state-of-the-art TLB design [303]

7.5.3. Fragmentation and Memory Bloat

When multiple concurrent GPGPU programs share the GPU, it is possible that a series of memory allocation and deallocation requests could create significant data fragmentation. While we do not observe this trend in any of the workloads we evaluate, as discussed in Section 7.3.2.1, Mosaic can potentially introduce data fragmentation and memory bloat. In order to perform additional analyses, we study the effect of pre-fragmenting the physical memory with random data, as described in Section 7.4, on performance.

In this study, we show the impact of data fragmentation and DRAM contention with Mosaic as a performance overhead compared to the baseline Mosaic with no pre-fragmented data. Additionally, we provide the memory footprint in terms of memory bloat after CAC performs all data movements during the compaction process throughout the execution of the workload. In Table 7.2, we pre-fragment a percentage of pages in main memory according to the fragmentation index, and within these fragmented large pages, we pre-allocate a percentage of base pages with some data that must be moved before coalescing. Then, we evaluate the performance of Mosaic with and without CAC, and provide the optimized CAC, which utilizes the in-DRAM bulk copy [64, 331]. any data movement.

Fragmentation Index	Large Page Occupancy	Memory Bloat (%)	Performance Overhead % (Optimized)	Improvement VS. no CAC
90%	1%	0.02%	0.03% (0.02%)	46.7%
90%	25%	0.06%	0.7% (0.7%)	45.6%
90%	50%	0.16%	5% (5%)	39.1%
95%	25%	0.39%	9.96% (9.9%)	33.4%
95%	50%	0.54%	12.6% (12.6%)	30.3%
97%	25%	0.75%	14.5% (14.4%)	24.3%
97%	50%	0.45%	17.1% (17.0%)	25.2%
100%	1%	10.66%	12.7% (5.3%)	30.2%
100%	10%	7.56%	26.5% (15.67%)	15.9%
100%	25%	7.20%	29.8% (21.7%)	13.0%
100%	35%	5.22%	30.6% (30.3%)	10.3%
100%	50%	3.37%	32.1% (30.4%)	11.0%
100%	75%	2.22%	33.3% (33.0%)	10.1%

Table 7.2. Performance comparison of Mosaic with various degree of fragmentation.

From Table 7.2, we make four conclusions. First, we conclude that CAC is effective in reducing the memory bloat, limiting the additional memory usage down to within 10.6% on average. Second, we found that the benefit of moving pages always outweighs the cost. Thus, we always observe performance improvement when CAC is applied, regardless of how fragmented the DRAM is. Third, we found that applying in-DRAM bulk copy can provide additional performance benefits when the physical DRAM fragmentation is sparse (i.e., high fragmentation index, but low page occupancy). This improvement comes from the fact that sparse DRAM increases the chance that CAC-optimized can find a target destination page that is either in the same subarray or in the same DRAM bank. Fourth, we do not observe any performance when fragmentation drops below 90%.

7.6. Mosaic: Conclusion

This dissertation explores the design space for transparent large page support for GPUs to sidestep the tradeoffs between TLB reach and demand paging latency. Our design, Mosaic, tracks temporal and spatial locality to inform policies for transparent coalescing of frequently accessed base pages and transparent splintering of large pages to reduce fragmentation, increase contiguity,

or reduce page far-fault latencies. Mosaic relies on techniques to preserve contiguity, allowing it to minimize synchronization and memory copy in the common case. A novel mechanism called LAZY-COALESCER allows application-level updates to virtual pages to proceed concurrently with coalescing or splintering, preserving atomicity for page table updates. Evaluation of a Mosaic prototype shows that using large pages lowers address translation overhead significantly without harming demand paging performance. Mosaic improves performance over TLB designs in the literature by 38.2%.

Chapter 8

Common Principles and Lesson Learned

This dissertation introduces several techniques that reduce memory interference in GPU-based systems. In this chapter, we provide a list of common design principles that are used throughout this dissertation as well as a summary of key lessons learned.

8.1. Common Design Principles

While techniques proposed in this dissertation are applied in different parts of the memory hierarchy, they share several key common principles. In this section, we reiterate over these common principles.

Identification of the Benefits of Threads from Using Shared Resources. The first common principle in this dissertation is to give shared resources only to threads that benefit from such shared resources. In many throughput processors, shared resources throughout the memory hierarchy are heavily contended due to the parallelism of these throughput processors. As a result, allowing all threads to freely use these shared resources usually leads to memory interference as we have analyzed thoroughly in Chapters 4, 5, 6 and 7. We observed that intelligently limiting the number of threads that use these shared resources often leads to significant performance improvement of GPU-based systems.

To this end, all mechanisms proposed in this dissertation modify shared resources such that

they 1) always prioritize threads that benefit from utilizing shared resources and 2) deprioritize threads that do not benefit from utilizing shared resources to avoid memory interference.

Division of Key Tasks of a Monolithic Structure into Simpler Structures. Another common principle that is utilized is the decoupling of key tasks on monolithic structures throughout the memory hierarchy. In MeDiC and MASK (See Chapters 4 and 6), we provide a mechanism that decouples the monolithic memory request buffer commonly used in modern systems into multiple queues, where different queues deal with different types of GPU memory requests. We found that the division of the monolithic request buffer simplifies the design of the memory scheduler. Specifically, it simplifies memory scheduler logic as the logic can now apply the same scheduling policy on each queue. A similar technique applies to SMS (See Chapter 5), which is a memory controller design for heterogeneous CPU-GPU systems.

8.2. Lessons Learned

This dissertation provides several techniques that together attempt to mitigate the performance impact of memory interference. While our analysis and evaluation have shown that our proposed techniques are effective in reducing the memory interference on various types of GPU-based systems, this dissertation also provides two important lessons. In this section, we summarize these two major lessons learned from our analysis.

Memory Latency is Important for the Performance of Throughput Processors. Typically, limited off-chip memory bandwidth is the major performance bottleneck of throughput processors. In this work, we show that the latency of memory requests also plays an important role in increasing the performance of throughput processors. First, we show that it is possible to reduce the number of cycles many warps are stalled by prioritizing the slowest thread within each warp. Our techniques allow these slow threads to benefit from the lower latency of the shared cache.

Second, we show that the memory latency of the page-walk-related requests is very important to the performance of GPU-based systems. In Chapters 6 and 7, we show that page walks can significantly reduces the memory hiding capability of GPU-based systems. As a result, it is crucial

to reduce the latency of these page-walk-related memory requests.

How to Design the GPU Memory Hierarchy to Avoid Memory Interference? This dissertation introduces several techniques across the main memory hierarchy of GPU-based systems. In this section, we provide recommended modifications for the memory hierarchy for both discrete GPUs as well as heterogeneous CPU-GPU systems.

The memory hierarchy of a discrete GPU should be designed to provide high throughput on both single-application and multi-application setups. As a result, the shared L2 data cache, the off-chip main memory and the shared TLB should be designed to minimize memory interference. To this end, MeDiC, MASK, and Mosaic (See Chapters 4, 6 and 7 for the detailed designs and analyses of these mechanisms) can be combined together to improve the efficiency of shared resources (the shared L2 cache, the shared TLB and the main memory). Specifically, we recommend system designers to modify the shared cache to 1) prioritize to threads that benefit from the shared L2 cache (e.g., threads from the *mostly-hit* and *all-hit* warp types), 2) deprioritize threads that are less likely to benefit from the shared L2 cache (e.g., threads from the *mostly-miss* and *all-miss* warp types), and 3) only cache page-walk-related data that would only benefit from using the shared data cache. Additionally, we recommend system designers to decouple the memory controller to perform two tasks hierarchically. The first task is to divide GPU memory request buffer into three different queues (*Golden*, *Silver* and *Normal* queues) similar to the design of MASK (See Section 6). To combine MASK with MeDiC, requests from the *mostly-hit* and *all-hit* warp types should be inserted into the *Silver Queue* to ensure that these requests have more priority than other data requests. Lastly, system designers should modify the GPU memory allocator to enforce the *soft guarantee* as defined in Section 7.3.2, which enables the GPU to provide low-overhead multi-page-size support.

To integrate our techniques into a CPU-GPU heterogeneous system, additional per-application FIFO queues can be integrated into the memory hierarchy as described in Section 5.3. This results in a memory hierarchy design that minimizes all types of memory interference that occur in GPU-based systems.

Chapter 9

Conclusions and Future Directions

In summary, the goal of this dissertation is to develop shared resource management mechanisms that can reduce memory interference in current and future throughput processors. To this end, we analyze memory interference that occurs in Graphics Processing Units, which are the prime example of throughput processors. Based on our analysis of GPU characteristics and the source of memory interference, we categorize memory interference into three different types: intra-application interference, inter-application interference and inter-address-space interference. We propose changes to the cache management and memory scheduling mechanisms to mitigate intra-application interference in GPGPU applications. We propose changes to the memory controller design and its scheduling policy to mitigate inter-application interference in heterogeneous CPU-GPU systems. We redesign the memory management unit and the memory hierarchy in GPUs to be aware of TLB-related data in order to mitigate the inter-address-space interference that originates from the address translation process. We introduce a hardware-software cooperative technique that modifies the memory allocation policy to enable large page support in order to further reduce the inter-address-space interference at the shared TLB. Our evaluations show that the GPU-aware cache and memory management techniques proposed in this dissertation are effective at mitigating the interference caused by GPUs on current and future GPU-based systems.

9.1. Future Research Directions

While this dissertation focuses on methods to mitigate memory interference in various GPU-based systems, this dissertation also uncovers new research topics. In this section, we describe potential research directions to further increase the performance of GPU-based systems.

9.1.1. Improving the Performance of the Memory Hierarchy in GPU-based Systems

Ways to Exploit Emerging High-Bandwidth Memory Technologies. 3D-stacked DRAM [155, 156, 157, 170, 213, 232] is an emerging main memory design that provides high bandwidth and high energy efficiency. We believe that analyzing how this new type of DRAM operates can expose techniques that might benefit modern GPU-based systems.

Aside from 3D-stacked memory, recent proposals provide methods to reduce DRAM latency [63, 199, 214, 215, 216], a method to utilize multi-ported DRAM [217], or methods to perform some computations within DRAM in order to reduce the amount of DRAM bandwidth [64, 151, 330, 331]. We think that these techniques, combined with observations on GPU applications' characteristics provided in this dissertation, can be applied to GPUs and should provide significant performance improvement for GPU-based systems.

Other Methods to Exploit Warp-type Heterogeneity and TLB-related Data in GPU-based system. In this dissertation, we show in Chapter 4 how GPU-based systems exploit warp-type heterogeneity to reduce *intra-application* interference and improve the effectiveness of the cache and the main memory. We also show in Chapter 6 how to design a GPU memory hierarchy that is aware of TLB data to minimize *inter-address-space* interference. We believe that it is beneficial to integrate these warp-type and TLB-awareness characteristics to the memory hierarchy in GPU-based systems to further improve system performance.

Potential Denial-of-service in Software Managed Shared Memory. Allowing GPU-based systems to be shared across multiple GPGPU applications potentially introduces new performance bottlenecks. Concurrently running multiple GPU applications creates a unique resource contention at GPU's software-managed Shared Scratchpad Memory. Because this particular resource is man-

aged by the GPGPU applications (in software), GPGPU applications that share the GPU all contend for this resource. The lack of communication between each GPGPU application prevents one application to inform its demand for the Shared Scratchpad Memory to other applications. As a result, one application can completely block other applications by using *all* Shared Scratchpad Memory.

It is possible to solve this unique problem through modifications in the hypervisor. For example, additional kernel scheduling techniques can be applied to 1) probe how much Shared Scratchpad Memory is needed by each application and 2) enforce a proper policy that only grants each application a portion of the Shared Scratchpad Memory.

Interference Management in GPUs for Emerging Applications. The emergence of embedded applications introduces a new requirement: real-time deadlines. Traditionally, these applications run on an embedded device which contains multiple application-specific integrated circuits (ASICs) to handle most of the computations. However, the rise of integrated GPUs in modern System-on-Chips (e.g., [71, 249, 268, 269]) as well as better GPU support in several cloud infrastructures (e.g., [28, 29, 368, 381]) allow these applications to perform these computations on the GPUs. While the GPUs can provide good IPC throughput due to their parallelism, the GPUs and the GPUs' memory hierarchy, also need to provide a low response time, or in many cases enforce *hard* performance guarantees (i.e., an application must finish its execution within a certain time limit).

Even though mechanisms proposed in this dissertation aim to minimize the slowdown caused by interference, these mechanisms do not provide actual performance guarantees. However, we believe it is possible to use observations in this dissertation to aid in designing mechanisms to provide a *hard* performance guarantee and limit the amount of memory interference when multiple of these new embedded applications are concurrently sharing GPU-based systems.

9.1.2. Low-overhead Virtualization Support in GPU-based Systems

While this dissertation proposes mechanisms to minimize inter-address-space interference in GPU-based systems, there are several open-ended research questions on how to efficiently virtualize GPU-based systems and how to efficiently shared other non-memory resources across multiple applications.

Maintaining Virtual Address Space Contiguity. While Chapter 7 provides a mechanism that maintains contiguous physical address, Mosaic does not perform compaction in the virtual address space as this dissertation does not observe virtual address space fragmentation in current GPGPU applications. However, it might be possible that a long chain of small size memory allocations and deallocations can break contiguity within the virtual address space. In this case, the virtual address space has to be remapped in order to create a contiguous chunk of unallocated virtual memory. This can lower the performance of GPU-based systems.

Utilizing High-bandwidth Interconnects to Transfer Data between CPU Memory and GPU Memory. As shown in Chapter 7, demand paging can be costly, especially when a large amount of data has to be transferred to the GPU. The long latency of demand paging can lead to significant stall time for GPU cores. Methods to improve the performance of demand paging remain a potential research problem. Emerging technologies such as NVIDIA's NVLink [108] and AMD's Infinity [71] can improve the data transfer rate between the CPUs and the GPUs. However, there is a lack of details on how to integrate these high-bandwidth interconnects to existing GPU hardware. Analyzing how these technologies operate, and providing a detailed study of their potential benefits and limitations is crucial for the integration of these new technologies in GPU-based systems.

Aside from techniques that utilize new technologies, architectural techniques can also mitigate the long data transfer latency between CPU memory and GPU memory. We believe that methods such as preemptively fetching the data of potential pages or proactively evicting potentially unused data in GPU memory can be effective in reducing the performance impact of demand paging.

9.1.3. Providing an Optimal Method to Concurrently Execute GPGPU Applications

While this dissertation allows applications to share the GPUs more efficiently by limiting the memory interference, how to schedule kernels and how to map these kernels to GPU cores remain an open research problem. In this work, we assume 1) an equal partitioning of GPU cores for each GPGPU application, and 2) every application is scheduled to start at the same time. Because applications have a different amount of parallelism as well as bandwidth demand, the optimal number of GPU cores that should be assigned to each application varies not only across different applications, but also across different workload setups.

As a result, providing an optimal method to manage the execution of GPGPU applications on GPU-based systems is a very complex problem. However, we believe that using the knowledge of the resource demand of each application between system software and the GPU hardware can significantly reduce the complexity of the scheduler. Information such as the amount of thread-level parallelism, the expected amount of data parallelism, the expected memory usage, cache locality, memory locality, etc. can be used as hints to assist in providing desirable application-to-GPU-core mappings and kernel scheduling decisions. In this dissertation, we provide several observations regarding GPGPU applications' characteristics that might be useful for assisting the system software to provide better mapping and scheduling decisions (e.g., memory allocation behavior, warp characteristics).

9.2. Final Summary

We conclude and hope that this dissertation, with the analyses of memory interference and mechanisms to mitigate this memory interference, enables many new research directions that further improve the capability of GPU-based systems.

Other Contributions by the Author

During my Ph.D., I had opportunities to be involved in many other research projects. While these projects do not fit into the theme of this dissertation, they have helped me tremendously in learning an in-depth knowledge about the memory hierarchy as well as the GPU architecture. I would like to acknowledge these projects as well as my early works on Network-on-Chip (NoCs) that kicked start my Ph.D.

My interest in studying memory interference in the memory hierarchy starts from the interests in Network-on-Chip. I have an opportunity in collaborating with Kevin Chang and Chris Fallin on two power-efficient network-on-chip designs that focus on bufferless network-on-chip: HAT [61] and MinBD [100]. In addition, I have authored another work on a hierarchical bufferless network-on-chip design called HiRD [34, 35] and have released NOCulator, which is the simulation infrastructure for both MinBD and HiRD [1]. All these works focus on mechanisms to improve power efficiency and simplifying the design of NoCs without sacrificing system performance. I also have an opportunity collaborating with Reetuparna Das on another work called A2C [81], which studies the placement of applications to cores in NoCs. A2C allows operating systems to be able to place applications to cores in a way that minimize interference, which is also the main theme in this thesis.

In collaboration with Vivek Seshadri, I have worked on techniques to allow in-DRAM bulk copy called RowClone [331].

In collaboration with Donghyuk Lee, I have worked on a study that characterizes latency variation in DRAM cells and provides techniques to improve the performance of DRAM by incorpo-

rating latency variation [214]

In collaboration with Justin Meza and Hanbin Yoon, I have worked on techniques to manage resources for hybrid memory that consists of DRAM and Phased changed memory (PCM) [394].

In collaboration with Nandita Vijaykumar, I have worked on a technique that allows better utilization of GPU cores called CABA [378]. CABA uses a technique similar to helper threads in order to improve the utilization of GPUs.

In collaboration with Onur Kayiran and Gabriel H. Loh, I have worked on a technique that manages GPU concurrency in a heterogeneous architecture in order to reduce interference [191]. In addition, I also worked on a GPU power management technique that turns down datapath components that are not in the bottleneck [190].

Bibliography

[1] NOCulator. https://github.com/CMU-SAFARI/NOCulator, 2014.

[2] P. Abad et al. Rotary router: an efficient architecture for CMP interconnection networks. *ISCA*, 2007.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Man, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Vigas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, 2015.

[4] J. Adriaens, K. Compton, N. S. Kim, and M. Schulte. The Case for GPGPU Spatial Multitasking. In *HPCA*, 2012.

[5] Advanced Micro Devices. AMD Accelerated Processing Units.

[6] Advanced Micro Devices. AMD I/O Virtualization Technology (IOMMU) Specification.

[7] Advanced Micro Devices. AMD Radeon R9 290X. http://www.amd.com/us/press-releases/Pages/amd-radeon-r9-290x-2013oct24.aspx.

[8] Advanced Micro Devices. ATI Radeon GPGPUs. *http://www.amd.com/us/products/desktop/graphics/amd-radeon-hd-6000/Pages/amd-radeon-hd-6000.aspx*.

[9] Advanced Micro Devices. OpenCL: The Future of Accelerated Application Performance Is Now.

[10] Advanced Micro Devices. *AMD-V Nested Paging*, 2010. http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf.

[11] Advanced Micro Devices. AMD Graphics Cores Next (GCN) Architecture. http://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf, 2012.

[12] Advanced Micro Devices. Heterogeneous System Architecture: A Technical Review. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/hsa10.pdf, 2012.

[13] Advanced Micro Devices. What is Heterogeneous System Architecture (HSA)?, 2013.

[14] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. Technical report, Cambridge, MA, USA, 1991.

[15] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch. Unlocking Bandwidth for GPUs in CC-NUMA Systems. In *HPCA*, 2015.

[16] A. Agrawal, A. Ansari, and J. Torrellas. Mosaic: Exploiting the Spatial Locality of Process Variation to Reduce Refresh Energy in On-chip eDRAM Modules. In *HPCA*, 2014.

[17] A. Agrawal, M. O'Connor, E. Bolotin, N. Chatterjee, J. Emer, and S. Keckler. CLARA: Circular Linked-List Auto and Self Refresh Architecture. In *MEMSYS*, 2016.

[18] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *ISCA*, 2015.

[19] J. Ahn, S. Jin, and J. Huh. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. In *ISCA*, 2012.

[20] J. Ahn, S. Jin, and J. Huh. Fast Two-Level Address Translation for Virtualized Systems. In *IEEE TC*, 2015.

[21] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *ISCA*, 2015.

[22] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber. Improving System Energy Efficiency with Memory Rank Subsetting. *ACM TACO*, 9(1):4:1–4:28, 2012.

[23] J. H. Ahn, J. Leverich, R. Schreiber, and N. P. Jouppi. Multicore DIMM: an Energy Efficient Memory Module with Independently Controlled DRAMs. *IEEE CAL*, 2009.

[24] B. Akin, F. Franchetti, and J. C. Hoe. Data Reorganization in Memory Using 3D-stacked DRAM. In *ISCA*, 2015.

[25] A. R. Alameldeen and D. A. Wood. Interactions Between Compression and Prefetching in Chip Multiprocessors. In *HPCA*, 2007.

[26] J. B. Alex Chen and X. Amatriain. Distributed Neural Networks with GPUs in the AWS cloud. 2014.

[27] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *ICS*, 1990.

[28] Amazon. Amazon EC2 GPU Instance. *http://aws.amazon.com/about-aws/whats-new/2013/11/04/announcing-new-amazon-ec2-gpu-instance-type/*.

[29] Amazon. An Introduction to High Performance Computing on AWS. `https://d0.awsstatic.com/whitepapers/Intro_to_HPC_on_AWS.pdf`, 2015.

[30] N. Amit, M. Ben-Yehuda, and B.-A. Yassour. IOMMU: Strategies for Mitigating the IOTLB Bottleneck. In *ISCA*, 2012.

[31] Apple Inc. *Huge Page Support in Mac OS X*. [Accessed April-2017].

[32] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, and C.-J. Wu. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *ISCA*, 2017.

[33] R. Ausavarungnirun, K. Chang, L. Subramanian, G. Loh, and O. Mutlu. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *ISCA*, 2012.

[34] R. Ausavarungnirun, C. Fallin, X. Yu, K. Chang, G. Nazario, R. Das, G. H. Loh, and O. Mutlu. Design and Evaluation of Hierarchical Rings with Deflection Routing. In *SBAC-PAD*, 2014.

[35] R. Ausavarungnirun, C. Fallin, X. Yu, K. Chang, G. Nazario, R. Das, G. H. Loh, and O. Mutlu. A Case for Hierarchical Rings with Deflection Routing. *PARCO*, 54(C):29–45, May 2016.

[36] R. Ausavarungnirun, S. Ghose, O. Kayran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. In *PACT*, 2015.

[37] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the Problems and Opportunities Posed by Multiple On-chip Memory Controllers. In *PACT*, 2010.

[38] O. O. Babarinsa and S. Idreos. JAFAR: Near-Data Processing for Databases. In *SIGMOD*, 2015.

[39] S. Baek, S. Cho, and R. Melhem. Refresh Now and Then. *IEEE TC*, 63(12):3114–3126, 2014.

[40] J.-L. Baer and T.-F. Chen. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE TC*, 44(5):609–623, 1995.

[41] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.

[42] P. Baran. On Distributed Communications Networks. 1964.

[43] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The Illiac IV Computer. *IEEE TC*, 100(8):746–757, 1968.

[44] T. W. Barr, A. L. Cox, and S. Rixner. Translation Caching: Skip, Don'T Walk (the Page Table). In *ISCA*, 2010.

[45] T. W. Barr, A. L. Cox, and S. Rixner. SpecTLB: A Mechanism for Speculative Address Translation. In *ISCA*, 2011.

[46] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient Virtual Memory for Big Memory Servers. In *ISCA*, 2013.

[47] I. Bhati, Z. Chishti, S.-L. Lu, and B. Jacob. Flexible Auto-refresh: Enabling Scalable and Energy-efficient DRAM Refresh Reductions. In *ISCA*, 2015.

[48] A. Bhattacharjee. Large-reach Memory Management Unit Caches. In *MICRO*, 2013.

[49] A. Bhattacharjee, D. Lustig, and M. Martonosi. Shared Last-level TLBs for Chip Multiprocessors. In *HPCA*, 2011.

[50] A. Bhattacharjee and M. Martonosi. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In *PACT*, 2009.

[51] A. Bhattacharjee and M. Martonosi. Inter-core Cooperative TLB for Chip Multiprocessors. In *ASPLOS*, 2010.

[52] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation Lookaside Buffer Consistency: A Software Approach. In *ASPLOS*, 1989.

[53] A. Boroumand, S. Ghose, B. Lucia, K. Hsieh, K. Malladi, H. Zheng, and O. Mutlu. LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. *IEEE CAL*, 2016.

[54] D. Bouvier and B. Sander. Applying AMD's "Kaveri" APU for Heterogeneous Computing. 2014.

[55] B. Burgess, B. Cohen, J. Dundas, J. Rupley, D. Kaplan, and M. Denman. Bobcat: AMD's Low-Power x86 Processor. *IEEE Micro*, 2011.

[56] M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. In *IISWC*, 2012.

[57] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *SIGMETRICS*, 1995.

[58] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory Controller. In *HPCA*, 1999.

[59] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an Embedded Data Parallel Language. In *SIGPLAN*, 2011.

[60] K. Chandrasekar, S. Goossens, C. Weis, M. Koedam, B. Akesson, N. Wehn, and K. Goossens. Exploiting Expendable Process-Margins in DRAMs for Run-Time Performance Optimization. In *DATE*, 2014.

[61] K. Chang, R. Ausavarungnirun, C. Fallin, and O. Mutlu. HAT: Heterogeneous Adaptive Throttling for On-Chip Networks. In *SBAC-PAD*, 2012.

[62] K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. In *SIGMETRICS*, 2016.

[63] K. Chang, D. Lee, Z. Chishti, A. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu. Improving DRAM Performance by Parallelizing Refreshes with Accesses . In *HPCA*, 2014.

[64] K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu. Low-cost Inter-linked Subarrays (LISA): Enabling Fast Inter-subarray Data Movement in DRAM. In *HPCA*, 2016.

[65] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian. Managing DRAM Latency Divergence in Irregular GPGPU Applications. In *SC*, 2014.

[66] N. Chatterjee, M. Shevgoor, R. Balasubramonian, A. Davis, Z. Fang, R. Illikkal, and R. Iyer. Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access. In *MICRO*, 2012.

[67] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman. Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches. In *PACT*, 2012.

[68] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*, 2009.

[69] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W. W. Hwu. Adaptive Cache Management for Energy-Efficient GPU Computing. In *MICRO*, 2014.

[70] X. Chen, S. Wu, L.-W. Chang, W.-S. Huang, C. Pearson, Z. Wang, and W. W. Hwu. Adaptive Cache Bypass and Insertion for Many-Core Accelerators. In *MES*, 2014.

[71] M. Clark. A New X86 Core Architecture for the Next Generation of Computing. In *HotChips*, 2016.

[72] J. D. Collins and D. M. Tullsen. Hardware Identification of Cache Conflict Misses. In *MICRO*, 1999.

[73] J. Cong, Z. Fang, Y. Hao, and G. Reinmana. Supporting Address Translation for Accelerator-Centric Architectures. In *HPCA*, 2017.

[74] Control Data Corporation. Control Data 7600 Computer Systems Reference Manual, 1972.

[75] R. Cooksey, S. Jourdan, and D. Grunwald. A Stateless, Content-directed Data Prefetching Mechanism. In *ASPLOS*, 2002.

[76] Couchbase Inc. Often Overlooked Linux OS Tweaks. [Accessed March, 2014].

[77] B. A. Crane and J. A. Githens. Bulk Processing in Distributed Logic Memory. *IEEE EC*, 14(2):186–196, April 1965.

[78] F. Dahlgren, M. Dubois, and P. Stenström. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. *IEEE TPDS*, 6(7):733–746, 1995.

[79] H. Dai, C. Li, H. Zhou, S. Gupta, C. Kartsaklis, and M. Mantor. A Model-driven Approach to Warp/thread-block Level GPU Cache Bypassing. In *DAC*, 2016.

[80] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *GPGPU*, 2010.

[81] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi. Application-to-core Mapping Policies to Reduce Memory System Interference in Multi-core Systems. In *HPCA*, 2013.

[82] R. Das, S. Eachempati, A. K. Mishra, V. Narayanan, and C. R. Das. Design and Evaluation of Hierarchical On-Chip Network Topologies for Next Generation CMPs. *HPCA*, 2009.

[83] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das. Application-aware Prioritization Mechanisms for On-chip Networks. In *MICRO*, 2009.

[84] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das. Aérgia: Exploiting Packet Latency Slack in On-chip Networks. In *ISCA*, 2010.

[85] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca. The Architecture of the DIVA Processing-in-memory Chip. In *ICS*, 2002.

[86] Y. Du, M. Zhou, B. Childers, D. Mosse, and R. Melhem. Supporting Superpages in Non-contiguous Physical Memory. In *HPCA*, 2015.

[87] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Orti. rCUDA: Reducing the Number of GPU-based Accelerators in High Performance Clusters. In *HPCS*, 2010.

[88] T. H. Dunigan. Kendall Square Multiprocessor: Early Experiences and Performance. In *of the Intel Paragon, ORNL/TM-12194*, 1994.

[89] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum. Improving Cache Management Policies Using Dynamic Reuse Distances. In *MICRO*, 2012.

[90] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In *ASPLOS*, 2010.

[91] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Prefetch-aware Shared Resource Management for Multi-core Systems. In *ISCA*, 2011.

[92] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. *ACM TOCS*, 30(7), 2012.

[93] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt. Parallel Application Memory Scheduling. In *MICRO*, 2011.

[94] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated Control of Multiple Prefetchers in Multi-core Systems. In *MICRO*, 2009.

[95] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for Bandwidth-efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems. In *HPCA*, 2009.

[96] Y. Etsion and D. G. Feitelson. Exploiting Core Working Sets to Filter the L1 Cache with Random Sampling. *IEEE TC*, 61(11):1535–1550, 2012.

[97] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3), 2008.

[98] S. Eyerman and L. Eeckhout. Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance. *IEEE CAL*, 2014.

[99] C. Fallin, C. Craik, and O. Mutlu. CHIPPER: A Low-complexity bufferless deflection router. In *HPCA*, 2011.

[100] C. Fallin, G. Nazario, X. Yu, K. Chang, R. Ausavarungnirun, and O. Mutlu. MinBD: Minimally-Buffered Deflection Routing for Energy-Efficient Interconnect. In *NoCs*, 2012.

[101] C. Fallin, G. Nazario, X. Yu, K. Chang, R. Ausavarungnirun, and O. Mutlu. *Bufferless and Minimally-Buffered Deflection Routing, in Routing Algorithms in Networks-on-Chip*, pages 241–275. Springer New York, New York, NY, 2014.

[102] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules. In *HPCA*, 2015.

[103] M. Fattah et al. A Low-Overhead, Fully-Distributed, Guaranteed-Delivery Routing Algorithm for Faulty Network-on-Chips. In *NOCS*, 2015.

[104] M. Feng, C. Tian, and R. Gupta. Enhancing LRU Replacement via Phantom Associativity. In *INTERACT*, Feb 2012.

[105] J. A. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *ISCA*, 1983.

[106] M. Flynn. Very High-Speed Computing Systems. *Proc. of the IEEE*, 54(2), 1966.

[107] A. Fog. The Microarchitecture of Intel, AMD and VIA CPUs.

[108] D. Foley. Ultra-Performance Pascal GPU and NVLink Interconnect. In *HotChips*.

[109] B. B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas. Programming the FlexRAM Parallel Intelligent Memory System. In *PPoPP*, 2003.

[110] W. Fung, I. Sham, G. Yuan, and T. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO*, 2007.

[111] W. W. L. Fung and T. M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *HPCA*, 2011.

[112] J. Gandhi, , M. D. Hill, and M. M. Swift. Exceeding the Best of Nested and Shadow Paging. In *ISCA*, 2016.

[113] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift. Efficient Memory Virtualization. In *MICRO*, 2014.

[114] H. Gao and C. Wilkerson. A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing. In *JWAC*, 2010.

[115] M. Gao, G. Ayers, and C. Kozyrakis. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *PACT*, 2015.

[116] M. Gao and C. Kozyrakis. HRL: Efficient and Flexible Reconfigurable Logic for Near-data Processing. In *HPCA*, 2016.

[117] J. Gaur, M. Chaudhuri, and S. Subramoney. Bypass and Insertion Algorithms for Exclusive Last-Level Caches. In *ISCA*, 2011.

[118] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-Efficient Mechanisms for Managing Thread Context in Throughput Processors. In *ISCA*, 2011.

[119] S. Ghose, H. Lee, and J. F. Martínez. Improving Memory Scheduling via Processor-side Load Criticality Information. In *ISCA*, 2013.

[120] M. Gokhale, B. Holmes, and K. Iobst. Processing in Memory: the Terasys Massively Parallel PIM Array. *Computer*, 28(4):23–31, 1995.

[121] C. Gómez, M. Gómez, P. López, and J. Duato. Reducing Packet Dropping in a Bufferless NoC. *EuroPar*, 2008.

[122] M. Gorman and P. Healy. Supporting Superpage Allocation Without Additional Hardware Support. In *ISMM*, 2008.

[123] M. Gorman and P. Healy. Performance Characteristics of Explicit Superpage Support. In *WIOSCA*, 2010.

[124] N. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A Memory Model for Scientific Algorithms on Graphics Processors. In *SC*, 2006.

[125] J. D. Grimes, L. Kohn, and R. Bharadhwaj. The Intel i860 64-bit Processor: A General-purpose CPU with 3D Graphics Capabilities. *IEEE CGA*, 9(4):85–94, 1989.

[126] R. Grindley, T. Abdelrahman, S. Brown, S. Caranci, D. DeVries, B. Gamsa, A. Grbic, M. Gusat, R. Ho, O. Krieger, et al. The NUMAchine Multiprocessor. In *ICPP*, 2000.

[127] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu. Express Cube Topologies for On-Chip Interconnects. In *HPCA*, 2009.

[128] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu. Kilo-NOC: A Heterogeneous Network-on-chip Architecture for Scalability and Service Guarantees. In *ISCA*, 2011.

[129] B. Grot, S. Keckler, and O. Mutlu. Topology-aware Quality-of-service Support in Highly Integrated Chip Multiprocessors. In *WIOSCA*, 2010.

[130] B. Grot, S. W. Keckler, and O. Mutlu. Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-Chip. In *MICRO*, 2009.

[131] M. Gschwind. Chip Multiprocessing and the Cell Broadband Engine. In *CF*, 2006.

[132] J. Gummaraju, M. Erez, J. Coburn, M. Rosenblum, and W. J. Dally. Architectural Support for the Stream Execution Model on General-Purpose Processors. In *PACT*, 2007.

[133] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T.-M. Low, L. Pileggi, J. C. Hoe, and F. Franchetti. 3D-Stacked Memory-Side Acceleration: Accelerator and System Design. In *WONDP*, 2014.

[134] S. Gupta, H. Gao, and H. Zhou. Adaptive Cache Bypassing for Inclusive Last Level Caches. In *IPDPS*, 2013.

[135] D. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, 1992.

[136] R. H. Halstead and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *ISCA*, 1988.

[137] V. C. Hamacher and H. Jiang. Hierarchical Ring Network Configuration and Performance Modeling. *IEEE TC*, 2001.

[138] T. D. Han and T. S. Abdelrahman. Reducing Branch Divergence in GPU Programs. In *GPGPU*, 2011.

[139] C. A. Hart. CDRAM in a Unified Memory Architecture. In *Intl. Computer Conference*, 1994.

[140] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt. Accelerating Dependent Cache Misses with an Enhanced Memory Controller. In *ISCA*, 2016.

[141] M. Hashemi, O. Mutlu, and Y. N. Patt. Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads. In *MICRO*, 2016.

[142] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu. ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality. In *HPCA*, 2016.

[143] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu. SoftMC: A Flexible and Practical Open-source Infrastructure for Enabling Experimental DRAM Studies. In *HPCA*, 2017.

[144] M. Hayenga, N. E. Jerger, and M. Lipasti. SCARAB: A Single Cycle Adaptive Routing and Bufferless Network. In *MICRO*, 2009.

[145] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *PACT*, 2008.

[146] H. Hellerman. Parallel Processing of Algebraic Expressions. *IEEE Transactions on Electronic Computers*, EC-15(1):82–91, Feb 1966.

[147] A. Herrera. NVIDIA GRID: Graphics Accelerated VDI with the Visual Performance of a Workstation. May 2014.

[148] H. Hidaka, Y. Matsuda, M. Asakura, and K. Fujishima. The Cache DRAM Architecture. *IEEE Micro*, 1990.

[149] W. Hillis. *The Connection Machine*. MIT Press, 1989.

[150] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. In *ISCA*, 2009.

[151] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. Transparent Offloading and Mapping (TOM): Enabling Programmer-transparent Near-data Processing in GPU Systems. In *ISCA*, 2016.

[152] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu. Accelerating Pointer Chasing in 3D-stacked Memory: Challenges, Mechanisms, Evaluation. In *ICCD*, 2016.

[153] W.-C. Hsu and J. E. Smith. Performance of Cached DRAM Organizations in Vector Supercomputers. In *ISCA*, 1993.

[154] I. Hur and C. Lin. Memory Prefetching Using Adaptive Stream Detection. In *MICRO*, 2006.

[155] Hybrid Memoty Cube Consortium. High-Bandwidth Memory White Paper.

[156] Hybrid Memoty Cube Consortium. HMC Specification 1.1, 2013.

[157] Hybrid Memoty Cube Consortium. HMC Specification 2.0, 2014.

[158] T. Ikeda and K. Kise. Application Aware DRAM Bank Partitioning in CMP. In *ICPADS*, 2013.

[159] Intel Corp. Intel®I/O Acceleration Technology. `http://www.intel.com/content/www/us/en/wireless-network/accel-technology.html`.

[160] Intel Corporation. Intel virtualization technology for directed i/o.

[161] Intel Corporation. Sandy Bridge Intel Processor Graphics Performance Developer's Guide.

[162] Intel Corporation. Intel architecture mmx technology in business applications. 1997. `http://download.intel.com/design/PentiumII/papers/24336702.PDF`.

[163] Intel Corporation. Products (Formerly Ivy Bridge), 2012.

[164] Intel Corporation. Introduction to intel architecture. 2014. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introduction-basics-paper.pdf`.

[165] Intel Corporation. Intel 64 and ia-32 architectures software developers manual. 2016. `https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`.

[166] Intel Corporation. 6th generation intel core processor family datasheet, vol. 1. 2017. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/desktop-6th-gen-core-family-datasheet-vol-1.pdf`.

[167] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive Insertion Policies for Managing Shared Caches. In *PACT*, 2008.

[168] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *ISCA*, 2010.

[169] J. Jalminger and P. Stenstrom. A Novel Approach to Cache Block Reuse Predictions. In *ICPP*, 2003.

[170] JEDEC. High Bandwidth Memory (HBM), 2013.

[171] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver. A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC. In *DAC*, 2012.

[172] W. Jia, K. A. Shaw, and M. Martonosi. MRPB: Memory Request Prioritization for Massively Parallel Processors. In *HPCA*, 2014.

[173] X. Jiang, Y. Solihin, L. Zhao, and R. Iyer. Architecture Support for Improving Bulk Memory Copying and Initialization Performance. In *PACT*, 2009.

[174] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das. Anatomy of GPU Memory System for Multi-Application Execution. In *MEMSYS*, 2015.

[175] A. Jog, O. Kayıran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Orchestrated Scheduling and Prefetching for GPGPUs. In *ISCA*, 2013.

[176] A. Jog, O. Kayıran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *ASPLOS*, 2013.

[177] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Exploiting Core Criticality for Enhanced GPU Performance. In *SIGMETRICS*, 2016.

[178] L. K. John and A. Subramanian. Design and Performance Evaluation of A Cache Assist to Implement Selective Caching. In *ICCD*, 1997.

[179] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In *ISCA*, 1997.

[180] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *ISCA*, 1990.

[181] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM JRD*, 2005.

[182] G. B. Kandiraju and A. Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-driven Study. In *ISCA*, 2002.

[183] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. In *ICCD*, 1999.

[184] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal. Redundant Memory Mappings for Fast Access to Large Memories. In *ISCA*, 2015.

[185] I. Karlin, A. Bhatele, J. Keasler, B. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *IPDPS*, 2013.

[186] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 Updates and Changes. 2013.

[187] D. Kaseridis, J. Stuecheli, and L. K. John. Minimalist Open-page: A DRAM Page-mode Scheduling Policy for the Many-core Era. In *MICRO*, 2011.

[188] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-Class GPU Resource Management in the Operating System. In *USENIX ATC*, 2012.

[189] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das. Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs. In *PACT*, 2013.

[190] O. Kayiran, A. Jog, A. Pattnaik, R. Ausavarungnirun, X. Tang, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das. uC-States: Fine-grained GPU Datapath Power Management. In *PACT*, 2016.

[191] O. Kayıran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das. Managing GPU Concurrency in Heterogeneous Architectures. In *MICRO*, 2014.

[192] G. Kedem and R. P. Koganti. WCDRAM: A Fully Associative Integrated Cached-DRAM with Wide Cache Lines. *CS-1997-03, Duke*, 1997.

[193] M. Kharbutli and Y. Solihin. Counter-Based Cache Replacement and Bypassing Algorithms. *IEEE TC*, 57(4):433–447, Apr. 2008.

[194] Khronos OpenCL Working Group. The OpenCL Specification. `http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf`, 2008.

[195] J. Kim and M. C. Papaefthymiou. Block-based Multi-period Refresh for Energy Efficient Dynamic Memory. In *ASIC*, 2001.

[196] K. Kim and J. Lee. A New Investigation of Data Retention Time in Truly Nanoscaled DRAMs. In *EDL*, 2009.

[197] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA*, 2010.

[198] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *MICRO*, 2010.

[199] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. In *ISCA*, 2012.

[200] A. K. Kodi, A. Sarathy, and A. Louri. iDEAL: Inter-router Dual-function Energy and Area-efficient Links for Network-on-chip (NoC) Architectures. In *ISCA*, 2008.

[201] P. M. Kogge. EXECUBE-A New Architecture for Scaleable MPPs. In *ICPP*, 1994.

[202] S. Konstantinidou and L. Snyder. Chaos Router: Architecture and Performance. In *ISCA*, 1991.

[203] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating STT-RAM as an energy-efficient main memory alternative. In *ISPASS*, 2013.

[204] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel. Coordinated and Efficient Huge Page Management with Ingens. In *OSDI*, 2016.

[205] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block Prediction & Dead-block Correlating Prefetchers. In *ISCA*, 2001.

[206] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *ISCA*, 2009.

[207] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Phase Change Memory Architecture and the Quest for Scalability. *CACM*, 53(7):99–106, 2010.

[208] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-Change Technology and the Future of Main Memory. *IEEE Micro*, 30(1):143–143, 2010.

[209] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware DRAM Controllers. In *MICRO*, 2008.

[210] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware Memory Controllers. *IEEE TC*, 60(10):1406–1430, 2011.

[211] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt. DRAM-aware Last-level Cache Writeback: Reducing Write-caused Interference in Memory Systems. In *TR-HPS-2010-002*, April, 2010.

[212] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt. Improving Memory Bank-Level Parallelism in the Presence of Prefetching. In *MICRO*, 2009.

[213] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu. Simultaneous Multi-layer Access: Improving 3D-stacked Memory Bandwidth at Low Cost. *ACM TACO*, 12(4):63, 2016.

[214] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu. Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms. In *SIGMETRICS*, 2017.

[215] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu. Adaptive-latency DRAM: Optimizing DRAM Timing for the Common-case. In *HPCA*, 2015.

[216] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu. Tiered-latency DRAM: A Low Latency and Low Cost DRAM Architecture. In *HPCA*, 2013.

[217] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu. Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM. In *PACT*, 2015.

[218] S.-Y. Lee and C.-J. Wu. Characterizing GPU Latency Hiding Ability. In *ISPASS*, 2014.

[219] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *ISCA*, 2013.

[220] A. Li, G.-J. van den Braak, A. Kumar, and H. Corporaal. Adaptive and Transparent Cache Bypassing for GPUs. In *SC*, 2015.

[221] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou. Locality-Driven Dynamic GPU Cache Bypassing. In *ICS*, 2015.

[222] D. Li, M. Rhu, D. Johnson, M. O'Connor, M. Erez, D. Burger, D. Fussell, and S. Redder. Priority-Based Cache Allocation in Throughput Processors. In *HPCA*, 2015.

[223] T. Li, V. K. Narayana, and T. El-Ghazawi. Symbiotic Scheduling of Concurrent GPU Kernels for Performance and Energy Optimizations. In *CF*, 2014.

[224] Huge Pages Part 2 (Interfaces). `https://lwn.net/Articles/375096/`. [February, 2010].

[225] C. H. Lin, D. Y. Shen, Y. J. Chen, C. L. Yang, and M. Wang. SECRET: Selective Error Correction for Refresh Energy Reduction in DRAMs. In *ICCD*, 2012.

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

[227] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *MICRO*, 2008.

[228] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu. RAIDR: Retention-aware Intelligent DRAM Refresh. In *ISCA*, 2012.

[229] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems. In *PACT*, 2012.

[230] W. Liu, P. Huang, T. Kun, T. Lu, K. Zhou, C. Li, and X. He. LAMS: A Latency-aware Memory Scheduling Policy for Modern DRAM Systems. In *IPCCC*, 2016.

[231] W. Liu, W. Muller-Wittig, and B. Schmidt. Performance Predictions for General-Purpose Computation on GPUs. In *ICPP*, 2007.

[232] G. H. Loh. 3D-stacked Memory Architectures for Multi-core Processors. In *ISCA*, 2008.

[233] S.-L. Lu, Y.-C. Lin, and C.-L. Yang. Improving DRAM Latency with Dynamic Asymmetric Subarray. In *MICRO*, 2015.

[234] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[235] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu. Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory. In *DSN*, 2014.

[236] D. Lustig, A. Bhattacharjee, and M. Martonosi. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. *ACM TACO*, 2013.

[237] L. Ma and R. Chamberlain. A Performance Model for Memory Bandwidth Constrained Applications on Graphics Engines. In *ASAP*, 2012.

[238] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *ISCA*, 2000.

[239] M. Mao, W. Wen, X. Liu, J. Hu, D. Wang, Y. Chen, and H. Li. TEMP: Thread Batch Enabled Memory Partitioning for GPU. In *DAC*, 2016.

[240] C. McNairy and D. Soltis. Itanium 2 Processor Microarchitecture. *IEEE Micro*, 23(2):44–55, 2003.

[241] X. Mei and X. Chu. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE TPDS*, 28(1):72–86, Jan 2017.

[242] V. Mekkat, A. Holey, P.-C. Yew, and A. Zhai. Managing Shared Last-Level Cache in a Heterogeneous Multicore Processor. In *PACT*, 2013.

[243] J. Meng, D. Tarjan, and K. Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *ISCA*, 2010.

[244] J. Menon, M. de Kruijf, and K. Sankaralingam. iGPU: Exception Support and Speculative Execution on GPUs. In *ISCA*, 2012.

[245] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan. Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management. *IEEE CAL*, 2012.

[246] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu. A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory. In *WEED*, 2013.

[247] Micron Technology, Inc. 576Mb: x18, x36 RLDRAM3, 2011.

[248] Microsoft Corporation. *Large-Page Support in Windows*. [Accessed April-2017].

[249] R. Mijat. Take GPU Processing Power Beyond Graphics with Mali GPU Computing, 2012.

[250] A. K. Mishra, O. Mutlu, and C. R. Das. A Heterogeneous Multiple Network-on-chip Design: An Application-aware Approach. In *DAC*, 2013.

[251] MongoDB Inc. Disable Transparent Huge Pages (THP). [Accessed April, 2016].

[252] T. Moscibroda and O. Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-core Systems. In *USENIX Security*, 2007.

[253] T. Moscibroda and O. Mutlu. Distributed Order Scheduling and Its Application to Multi-core DRAM Controllers. In *PODC*, 2008.

[254] T. Moscibroda and O. Mutlu. A Case for Bufferless Routing in On-Chip Networks. In *ISCA*, 2009.

[255] J. Mukundan and J. F. Martinez. MORSE: Multi-objective Reconfigurable Self-optimizing Memory Scheduler. In *HPCA*, 2012.

[256] R. Mullins, A. West, and S. Moore. Low-latency Virtual-channel Routers for On-chip Networks. In *ISCA*, 2004.

[257] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In *MICRO*, 2011.

[258] O. Mutlu, H. Kim, and Y. N. Patt. Address-value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns. In *MICRO*, 2005.

[259] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for Efficient Processing in Runahead Execution Engines. In *ISCA*, 2005.

[260] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO*, 2007.

[261] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *ISCA*, 2008.

[262] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *HPCA*, 2003.

[263] P. J. Nair, D.-H. Kim, and M. K. Qureshi. ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates. In *ISCA*, 2013.

[264] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In *MICRO*, 2011.

[265] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, Transparent Operating System Support for Superpages. In *OSDI*, 2002.

[266] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An Adaptive Data Cache Prefetcher. In *PACT*, 2004.

[267] NuoDB Inc. Linux Transparent Huge Pages, JEMalloc and NuoDB. [Accessed May, 2014].

[268] NVIDIA Corporation. NVIDIA Tegra K1.

[269] NVIDIA Corporation. NVIDIA Tegra X1.

[270] NVIDIA Corporation. CUDA C/C++ SDK Code Samples. `http://developer.nvidia.com/cuda-cc-sdk-code-samples`, 2011.

[271] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. `http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf`, 2011.

[272] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2012.

[273] NVIDIA Corporation. NVIDIA GeForce GTX 750 Ti. 2014.

[274] NVIDIA Corporation. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2015.

[275] NVIDIA Corporation. Multi-Process Service. `https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`, 2015.

[276] NVIDIA Corporation. NVIDIA Tesla P100. `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`, 2016.

[277] NVIDIA Corporation. Parallel Thread Execution ISA Version 5.0. 2017.

[278] NVIDIA Corporation. Tuning CUDA Applications for Maxwell. 2017.

[279] G. Nychis, C. Fallin, T. Moscibroda, and O. Mutlu. Next Generation On-Chip Networks: What Kind of Congestion Control Do We Need? In *Hotnets*, 2010.

[280] G. Nychis, C. Fallin, T. Moscibroda, O. Mutlu, and S. Seshan. On-chip Networks from a Networking Perspective: Congestion and Scalability in Many-core Interconnects. In *SIGCOMM*, 2012.

[281] S. O, Y. H. Son, N. S. Kim, and J. H. Ahn. Row-Buffer Decoupling: A Case for Low-Latency DRAM Microarchitecture. In *ISCA*, 2014.

[282] T. Ohsawa, K. Kai, and K. Murakami. Optimizing the DRAM Refresh Count for Merged DRAM/Logic LSIs. In *ISLPED*, 1998.

[283] M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *ISCA*, 1998.

[284] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU Concurrency with Elastic Kernels. In *ASPLOS*, 2013.

[285] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective Superscalar Processors. In *ISCA*, 1997.

[286] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos. Prediction-based Superpage-friendly TLB Designs. In *HPCA*, 2015.

[287] J. Park, R. M. Yoo, D. S. Khudia, C. J. Hughes, and D. Kim. Location-aware Cache Management for Many-core Processors with Deep Cache Hierarchy. In *SC*, 2013.

[288] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In *MICRO*, 2004.

[289] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997.

[290] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *PACT*, 2016.

[291] PCI-SIG. PCI Express Base Specification Revision 3.1a, 2015.

[292] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler. A Case for Toggle-aware Compression for GPU Systems. In *HPCA*, 2016.

[293] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Exploiting Compressed Block Size as an Indicator of Future Reuse. In *HPCA*, 2015.

[294] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry. Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency. In *MICRO*, 2013.

[295] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Base-delta-immediate Compression: Practical Data Compression for On-chip Caches. In *PACT*, 2012.

[296] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):51–59, August 1996.

[297] Percona. Why TokuDB Hates Transparent HugePages. [Accessed July, 2014].

[298] S. Phadke and S. Narayanasamy. MLP Aware Heterogeneous Memory System. In *DATE*, 2011.

[299] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. Increasing TLB Reach by Exploiting Clustering in Page Translations. In *HPCA*, 2014.

[300] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. CoLT: Coalesced Large-Reach TLBs. In *MICRO*, 2012.

[301] B. Pham, J. Vesely, G. Loh, and A. Bhattacharjee. Large Pages and Lightweight Memory Management in Virtualized Systems: Can You Have it Both Ways? In *MICRO*, 2015.

[302] B. Pichai, L. Hsu, and A. Bhattacharjee. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *ASPLOS*, 2014.

[303] J. Power, M. D. Hill, and D. A. Wood. Supporting x86-64 Address Translation for 100s of GPU Lanes. In *HPCA*, 2014.

[304] PowerVR. PowerVR Hardware Architecture Overview for Developers. 2016. `http://cdn.imgtec.com/sdk-documentation/PowerVR+Hardware.Architecture+Overview+for+Developers.pdf`.

[305] T. Preis, P. Virnau, W. Paul, and J. J. Schneider. Accelerated Fluctuation Analysis by Graphic Cards and Complex Pattern Formation in Financial Markets. *New Journal of Physics*, 11, 2009.

[306] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. NDC: Analyzing the Impact of 3D-stacked Memory+logic Devices on MapReduce Workloads. In *ISPASS*, 2014.

[307] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive Insertion Policies for High Performance Caching. In *ISCA*, 2007.

[308] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling. In *MICRO*, 2009.

[309] M. K. Qureshi, D. H. Kim, S. Khan, P. J. Nair, and O. Mutlu. AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems. In *DSN*, 2015.

[310] M. K. Qureshi and Y. N. Patt. Utility-based Cache Partitioning: A Low-overhead, High-performance, Runtime Mechanism to Partition Shared Caches. In *MICRO*, 2006.

[311] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *ISCA*, 2009.

[312] B. R. Rau. Pseudo-randomly Interleaved Memory. In *ISCA*, 1991.

[313] G. Ravindran and M. Stumm. A Performance Comparison of Hierarchical Ring- and Mesh-connected Multiprocessor Networks. In *HPCA*, 1997.

[314] G. Ravindran and M. Stumm. On Topology and Bisection Bandwidth for Hierarchical-ring Networks for Shared Memory Multiprocessors. In *HPCA*, 1998.

[315] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education. In *WCAE*, 2004.

[316] Redis Labs. Redis Latency Problems Troubleshooting. [Accessed April, 2016].

[317] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA*, 2000.

[318] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *MICRO*, 2012.

[319] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-Aware Warp Scheduling. In *MICRO*, 2013.

[320] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy. UNified Instruction/Translation/Data (UNITD) Coherence: One Protocol to Rule them All. In *HPCA*, 2010.

[321] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *SOSP*, 2011.

[322] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *SIGOPS*, 2013.

[323] R. M. Russell. The CRAY-1 Computer System. *CACM*, 21(1):63–72, 1978.

[324] Y. Sato, T. Suzuki, T. Aikawa, S. Fujioka, W. Fujieda, H. Kobayashi, H. Ikeda, T. Nagasawa, A. Funyu, Y. Fuji, K. Kawasaki, M. Yamazaki, and M. Taguchi. Fast cycle RAM (FCRAM): A 20-ns Random Row Access, Pipe-Lined Operating DRAM. In *VLSIC*, 1998.

[325] A. Saulsbury, F. Dahlgren, and P. Stenström. Recency-based TLB Preloading. In *ISCA*, 2000.

[326] P. B. Schneck. *The CDC STAR-100*, pages 99–117. Springer US, Boston, MA, 1987.

[327] D. N. Senzig and R. V. Smith. Computer Organization for Array Processing. In *AFIPS*, 1965.

[328] S.-Y. Seo. Methods of Copying a Page in a Memory Device and Methods of Managing Pages in a Memory System. U.S. Patent Application 20140185395, 2014.

[329] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. The Dirty-Block Index. In *ISCA*, 2014.

[330] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. Kozuch, O. Mutlu, P. Gibbons, and T. Mowry. Fast Bulk Bitwise AND and OR in DRAM. *IEEE CAL*, 2015.

[331] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, et al. RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization. In *ISCA*, 2013.

[332] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM. In *arXiv CoRR*, 2016.

[333] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses. In *MICRO*, 2015.

[334] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry. The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing. In *PACT*, 2012.

[335] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks. *ACM TACO*, 11(4):51:1–51:22, 2015.

[336] T. Shanley. *Pentium Pro Processor System Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.

[337] W. Shin, J. Yang, J. Choi, and L.-S. Kim. NUAT: A Non-Uniform Access Time Memory Controller. In *HPCA*, 2014.

[338] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. In *PPoPP*, 2012.

[339] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi. A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch. In *MICRO*, 2012.

[340] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. Cache Coherence for GPU Architectures. In *HPCA*, 2013.

[341] SiSoftware. Benchmarks : Measuring GP (GPU/APU) Cache and Memory Latencies. `http://www.sisoftware.net`, 2014.

[342] R. L. Sites and R. T. Witek. *ALPHA Architecture Reference Manual*. Digital Press, Boston, Oxford, Melbourne, 1998.

[343] D. L. Slotnick, W. C. Borck, and R. C. McReynolds. The Solomon Computer – A Preliminary Report. In *Workshop on Computer Organization*, 1962.

[344] B. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE*, 1981.

[345] B. J. Smith. A Pipelined, Shared Resource MIMD Computer. In *ICPP*, 1978.

[346] Y. H. Son, S. O, Y. Ro, J. W. Lee, and J. H. Ahn. Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations. In *ISCA*, 2013.

[347] Splunk Inc. Transparent Huge Memory Pages and Splunk Performance. [Accessed December, 2013].

[348] S. Srikantaiah and M. Kandemir. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. In *MICRO*, 2010.

[349] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *HPCA*, 2007.

[350] H. S. Stone. A Logic-in-Memory Computer. *IEEE TC*, C-19(1):73–78, 1970.

[351] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, Mar. 2012.

[352] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John. The Virtual Write Queue: Coordinating DRAM and Last-level Cache Policies. In *ISCA*, 2010.

[353] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu. The Blacklisting Memory Scheduler: Achieving high performance and fairness at low cost. In *ICCD*, 2014.

[354] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu. The Blacklisting Memory Scheduler: Balancing Performance, Fairness and Complexity. *arXiv CoRR*, 2015.

[355] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu. BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling. In *IEEE TPDS*, 2016.

[356] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory. In *MICRO*, 2015.

[357] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In *HPCA*, 2013.

[358] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A Compiler Architecture for Performance-oriented Embedded Domain-specific Languages. In *TECS*, 2014.

[359] Z. Sura, A. Jacob, T. Chen, B. Rosenburg, O. Sallenave, C. Bertolli, S. Antao, J. Brunheroto, Y. Park, K. O'Brien, and R. Nair. Data Access Optimization in a Processing-in-memory System. In *CF*, 2015.

[360] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*, volume 1.

[361] Y. Suzuki, S. Kato, H. Yamada, and K. Kono. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? In *USENIX ATC*, 2014.

[362] Sybase Inc. SAP IQ and Linux Hugepages/Transparent Hugepages. [Accessed May, 2014].

[363] M. Talluri and M. D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *ASPLOS*, 1994.

[364] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling Preemptive Multiprogramming on GPUs. In *ISCA*, 2014.

[365] J. E. Thornton. Parallel Operation in the Control Data 6600. *AFIPS FJCC*, 1964.

[366] J. E. Thornton. Design of a Computerthe Control Data 6600. 1970.

[367] Transparent Hugepages. `https://lwn.net/Articles/359158/`. [October, 2009].

[368] K. Tian, Y. Dong, and D. Cowperthwaite. A Full GPU Virtualization Solution with Mediated Pass-Through. In *USENIX ATC*, 2014.

[369] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A Modified Approach to Data Cache Management. In *MICRO*, 1995.

[370] Univ. of British Columbia. GPGPU-Sim GTX 480 Configuration. `http://dev.ece.ubc.ca/projects/gpgpu-sim/browser/v3.x/configs/GTX480`.

[371] H. Usui, L. Subramanian, K. Chang, and O. Mutlu. SQUASH: Simple qos-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *arXiv CoRR*, 2015.

[372] H. Usui, L. Subramanian, K. Chang, and O. Mutlu. DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. *ACM TACO*, 12(4), Jan. 2016.

[373] H. Vandierendonck and A. Seznec. Fairness Metrics for Multi-threaded Processors. *IEEE CAL*, Feb 2011.

[374] R. Venkatesan, S. Herr, and E. Rotenberg. Retention-aware Placement in DRAM (RAPID): Software Methods for Quasi-non-volatile DRAM. In *HPCA*, 2006.

[375] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *ISPASS*, 2016.

[376] T. Vijayaraghavany, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, O. Kayiran, M. Meswani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan. Design and Analysis of an APU for Exascale Computing. In *HPCA*, 2017.

[377] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu. Zorua: A Holistic Approach to Resource Virtualization in GPUs. In *MICRO*, 2016.

[378] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu. A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps. In *ISCA*, 2015.

[379] Vivante. Vivante Vega GPGPU Technology. 2016. `http://www.vivantecorp.com/index.php/en/technology/gpgpu.html`.

[380] VoltDB Inc. VoltDB Documentation. [Accessed April, 2016].

[381] L. Vu, H. Sivaraman, and R. Bidarkar. GPU Virtualization for High Performance General Purpose Computing on the ESX Hypervisor. In *HPC*, 2014.

[382] Z. Wang, J. Yang, R. Melhem, B. R. Childers, Y. Zhang, and M. Guo. Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-Grained Sharing. In *HPCA*, 2016.

[383] F. A. Ware and C. Hampel. Improving Power and Data Efficiency with Threaded Memory Modules. In *ICCD*, 2006.

[384] S. Wasson. AMD's A8-3800 Fusion APU., Oct. 2011.

[385] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture Through Microbenchmarking. In *ISPASS*, 2010.

[386] Y. Wu, R. Rakvic, L.-L. Chen, C.-C. Miao, G. Chrysos, and J. Fang. Compiler Managed Micro-cache Bypassing for High Performance EPIC Processors. In *MICRO*, 2002.

[387] L. Xiang, T. Chen, Q. Shi, and W. Hu. Less Reused Filter: Improving L2 Cache Performance via Filtering Less Reused Lines. In *ICS*, 2009.

[388] P. Xiang, Y. Yang, and H. Zhou. Warp-Level Divergence in GPUs: Characterization, Impact, and Mitigation. In *HPCA*, 2014.

[389] M. Xie, D. Tong, K. Huang, and X. Cheng. Improving System Throughput and Fairness Simultaneously in Shared Memory CMP Systems via Dynamic Bank Partitioning. In *HPCA*, 2014.

[390] X. Xie, Y. Liang, G. Sun, and D. Chen. An Efficient Compiler Framework for Cache Bypassing on GPUs. In *ICCAD*, 2013.

[391] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. Coordinated Static and Dynamic Cache Bypassing for GPUs. In *HPCA*, 2015.

[392] D. Xiong, K. Huang, X. Jiang, and X. Yan. Memory Access Scheduling Based on Dynamic Multilevel Priority in Shared DRAM Systems. *ACM TACO*, 13(4), Dec. 2016.

[393] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming. In *ISCA*, 2016.

[394] H. Yoon, R. A. J. Meza, R. Harding, and O. Mutlu. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *ICCD*, 2012.

[395] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *ICCD*, 2012.

[396] B. Yu, J. Ma, T. Chen, and M. Wu. Global Priority Table for Last-Level Caches. In *DASC*, 2011.

[397] G. Yuan, A. Bakhoda, and T. Aamodt. Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures. In *MICRO*, 2009.

[398] C. Zhang, G. Sun, P. Li, T. Wang, D. Niu, and Y. Chen. SBAC: A Statistics Based Cache Bypassing Method for Asymmetric-access Caches. In *ISPLED*, 2014.

[399] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *HPDC*, 2014.

[400] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee. The Impulse Memory Controller. *IEEE TC*, 50(11):1117–1132, 2001.

[401] X. Zhang and Y. Yan. Comparative Modeling and Evaluation of CC-NUMA and COMA on Hierarchical Ring Architectures. *IEEE TPDS*, 1995.

[402] J. Zhao, O. Mutlu, and Y. Xie. FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems. In *MICRO*, 2014.

[403] L. Zhao, R. Iyer, S. Makineni, L. Bhuyan, and D. Newell. Hardware Support for Bulk Data Movement in Server Platforms. In *ICCD*, 2005.

[404] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu. Mini-rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency. In *MICRO*, 2008.

[405] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. Towards High Performance Paged Memory for GPUs. In *HPCA*, 2016.

[406] W. K. Zuravleff and T. Robinson. Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order. In *US Patent Number 5,630,096*, 1997.