Memory System Optimizations for CPU-GPU Heterogeneous Chip-multiprocessors

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

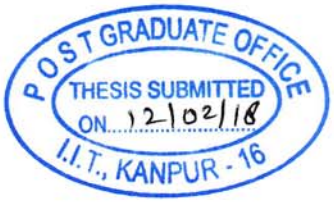
by

Siddharth Rai

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR, INDIA

July, 2018



CERTIFICATE

It is certified that the work contained in the thesis entitled "***Memory System Optimizations for CPU-GPU Heterogeneous Chip-multiprocessors***" by ***Siddharth Rai*** has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

February, 2018

Mainak Chaudhuri

Dr. Mainak Chaudhuri
Associate Professor,
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur,
Kanpur-208016, INDIA.

Synopsis

Recent commercial chip-multiprocessors (CMPs) have integrated CPU as well as GPU cores on the same chip [42, 43, 44, 93]. In today's designs, these cores typically share parts of the memory system resources between the applications executing on the two types of cores. However, since the CPU and the GPU cores execute very different workloads leading to very different resource requirements, designing intelligent protocols for sharing resources between them such that both CPU and GPU gain in performance brings forth new challenges to the design space of these heterogeneous processors. In this dissertation, we explore solutions to dynamically allocate last-level cache (LLC) capacity and DRAM bandwidth to the CPU and GPU cores in a design where both the CPU and the GPU share the large on-die LLC, DRAM controllers, DRAM channels, DRAM ranks, and DRAM device resources (banks, rows).

CPU and GPU differ vastly in their execution models, workload characteristics, and performance requirements. On one hand, a CPU core executes instructions of a latency-sensitive and/or moderately bandwidth-sensitive job progressively in a pipeline generating memory accesses (for instruction and data) only in a few pipeline stages (instruction fetch and data memory access stages). On the other hand, GPU can access different data streams having different semantic meanings and disparate

access patterns throughout the rendering pipeline. Such data streams include input vertex, pixel depth, pixel color, texture map, shader instructions, shader data (including shader register spills and fills), etc.. Without carefully designed shared resource management policies, the CPU and the GPU data streams may interfere with each other, leading to significant loss in CPU and GPU performance accompanied by degradation in GPU-rendered animation quality. However, in most workloads, the latency requirements of a GPU application are not as stringent as that of a CPU application. For example, when the GPU is used for 3D scene rendering, achieving a minimal frame rate that can deliver adequately satisfactory visual experience to the end user is enough. Similarly, for a GPGPU-style general-purpose application, long memory access latencies can be tolerated due to the massively parallel nature of the shader cores and a large number of ready-to-execute shader thread contexts.[1] The different computation models in the CPU and GPU cores open up scope for more efficient and novel management of the shared memory system resources in such architectures.

We divide the contributions of this dissertation into three parts. In the first part of the dissertation, we present an LLC management policy which dynamically estimates the reuse probabilities of the GPU streams as well as the CPU data by sampling portions of the CPU and GPU working sets and storing the sampled tags in a small working set sample cache. Since the GPU application working sets are typically very large, for this working set sample cache to be effective, it is custom-designed to have large footprint coverage while requiring only few tens of kilobytes of storage. We use the estimated reuse probabilities to design shared LLC policies

[1] These contexts are referred to as warps in Nvidia GPUs and this is the terminology we will use in this dissertation.

for handling hits and misses to reads and writes from both types of cores. We evaluate our proposal through simulation studies conducted on a detailed heterogeneous CMP simulator modeling one GPU and four CPU cores. The simulation results show that compared to a state-of-the-art baseline with a 16 MB shared LLC, our proposal can improve the performance (frame rate or execution cycles, as applicable) of eighteen GPU workloads spanning DirectX and OpenGL game titles as well as CUDA applications by 12% on average and up to 51%. The performance of the co-running quad-core CPU workload mixes improves by 7% on average and up to 19%.

In the second part of the dissertation, we present a memory system management policy driven by the quality of service (QoS) requirement of the GPU applications co-scheduled with the CPU applications on the heterogeneous CMP. Our proposal dynamically estimates the delivered level of QoS (e.g., frame rate in 3D scene rendering) of the GPU application. If the estimated QoS level meets the minimum acceptable QoS level, our proposal employs a light weight mechanism to dynamically adjust the GPU memory access rate so that the GPU is able to just meet the required QoS level. This frees up memory system resources which are then shifted to the co-running CPU workloads. Detailed simulations done on a heterogeneous CMP model with one GPU and four CPU cores running heterogeneous mixes of DirectX, OpenGL, and CPU applications show that our proposal improves the CPU performance by 18% on average.

The second contribution improves only those heterogeneous mixes where the GPU application already meets the required QoS level. The third part of the dissertation addresses the performance issues of the remaining heterogeneous workload mixes. In this part, we present a memory access scheduling algorithm driven by

the performance feedback from the integrated GPU. We observe that the streams sourced by different parts of the GPU pipeline do not respond equally to memory system resource allocations. To dynamically estimate this differing performance sensitivity of the streams, we propose a novel queuing network model which uses a set of saturating counters per rendering pipeline stage or sub-unit to generate criticality hints for different GPU access streams. If a GPU application is found to perform below the required QoS level, the memory scheduler uses this criticality information to partition DRAM bandwidth between the critical GPU accesses, non-critical GPU accesses, and CPU accesses, such that, the GPU performance improves without degrading the CPU performance by much. Detailed simulations done on a heterogeneous CMP model with one GPU and four CPU cores running heterogeneous mixes of DirectX, OpenGL, and CPU applications show that our proposal improves the GPU performance by 15% on average without degrading the CPU performance much. The GPGPU applications do not have any minimum QoS requirement. The goal is to improve the performance of these applications as much as possible. We propose extensions to our core mechanisms to handle the heterogeneous workload mixes containing GPGPU applications. These extensions improve the heterogeneous system performance by 7% on average for these mixes, where the heterogeneous system performance is defined as a performance metric that assigns equal weights to CPU and GPU performance.

In summary, the first contribution presents an effective shared LLC management policy for heterogeneous CMPs. The second contribution deals with both shared LLC management and DRAM access scheduling governed by a light-weight memory access throttling policy. The third contribution presents a DRAM access scheduling algorithm driven by the performance-criticality of GPU access streams. Taken

together, these three contributions present a holistic set of optimizations for the shared memory system resources of the emerging CPU-GPU heterogeneous CMPs.

Acknowledgements

The start and the completion of this dissertation would not have been possible without the help from my advisor, my friends, and my family.

Foremost, I am grateful to Prof. Mainak Chaudhuri for agreeing to advise me for the PhD. and helping me both intellectually and financially during this period.

Furthermore, having graduated from IIT Guwahati, I have been greatly influenced by Prof. Gautam Barua, whose inspiration has helped me move forward during this period. I would like to thank Prof. Gautam Barua for all the inspiration.

I would like to acknowledge the help I have received both before and during the PhD. from my colleagues at Oracle Bangalore and my friend Rohit Khanna through countless conversations both on academics and on other matters.

Lastly, I would like to thank my lab mates and my fellow PhD. student Sudhanshu Shukla for helping me through this period.

Siddharth Rai

Dedicated To

My Teachers, both at School and College

Contents

List of Tables

List of Algorithms

Chapter 1

Introduction

Microprocessor performance has seen incredible improvement over the last several decades [81]. Increased transistor count as per Moore's Law, and subsequent improvement in clock frequency and core count guided by Dennard Scaling have played a major role in this progress. Nonetheless, this progress has slowed down due to practical limits in transistor scaling and constraints imposed by the power budget in the last decade [20]. To overcome these challenges, heterogeneous processing has emerged as one of the new design paradigms for improving processor performance in an energy efficient manner [58]. In this design paradigm, different kinds of cores, optimized for single threaded (dynamically scheduled superscalar cores) and multi-threaded (single instruction multiple data / field-programmable gate arrays) performance are integrated on the same chip. In these heterogeneous processors, parts of the memory system resources such as LLC, interconnect, memory controllers, and DRAM banks are shared between the on-die cores to improve resource utilization [42, 43, 44, 93]. However, sharing these resources in a way such that all processing elements gain in performance brings forth new challenges in this design

space.

A single-chip CPU-GPU heterogeneous processor implements one such design, where parts of the memory system resources are shared between the latency optimized CPU and throughput optimized GPU cores. Moreover, when the GPU is used for a 3D scene rendering job, various fixed function units of the GPU (i.e., vertex processor, texture sampler, blitter, color writer, etc.) also contend for the memory system resources with the CPU. Today such processors can be found in the product lines of all major microprocessor vendors including AMD, Intel, and Nvidia [28, 42, 43, 44, 93]. These processors share a significant portion of the memory system resources between the CPU and GPU cores. For example, in AMD accelerated processing unit (APU) architectures [44], the CPU and GPU cores share everything beyond the on-die cache hierarchy including memory controllers and DRAM banks. In Intel's integrated GPU designs [42, 43, 93], the CPU and the GPU cores share the large on-die last-level (L3) cache in addition to sharing in-package L4 e-DRAM cache (available in Haswell, Broadwell, and Skylake parts), the on-die interconnect, memory controllers, and the DRAM banks. Similar to AMD APU designs, Nvidia's Tegra [28] series of mobile processor integrates CPU and GPU cores sharing an on-die memory controller.

In such a tightly integrated system, when both CPU and GPU execute jobs at the same time, performance of both the processors degrades due to destructive interference caused by the shared resource contention. To understand the extent of CPU-GPU interference, we conducted a set of experiments on an Intel Core i7-4770K processor (Haswell)-based platform. This heterogeneous processor has four CPU cores and an integrated Intel HD 4600 GPU sharing an 8 MB LLC and dual-channel DDR3-1600 16 GB DRAM (25.6 GB/s peak DRAM bandwidth). We pre-

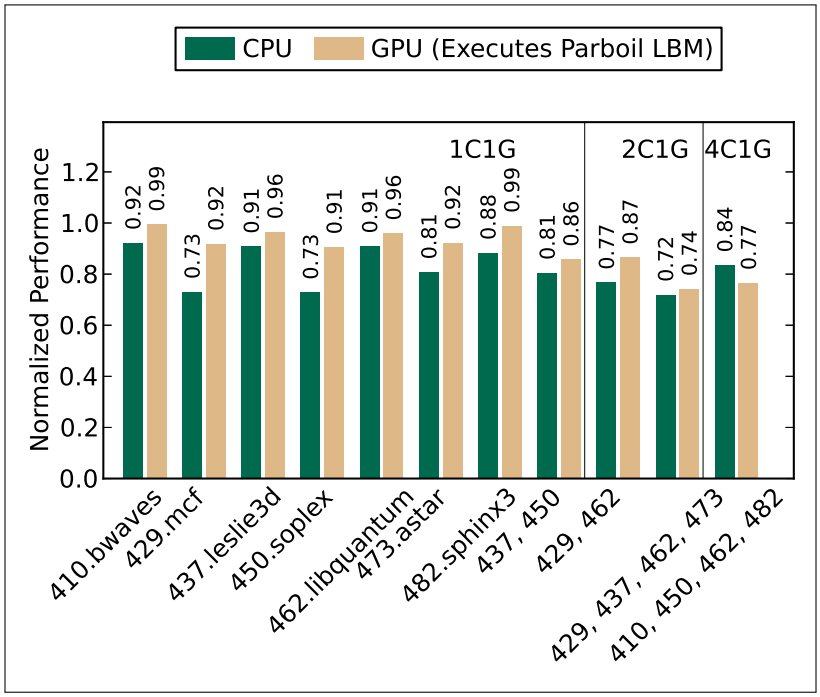


Figure 1.1: Performance of heterogeneous mixes relative to standalone performance on Core i7-4770K.

pared eleven heterogeneous mixes for these experiments. Every mix has LBM from the Parboil OpenCL suite [97] as the GPU workload using the long input (LBM serves as a representative for memory-intensive GPU workloads). Seven out of the eleven mixes exercise one CPU core, two mixes exercise two CPU cores, and the remaining two mixes exercise all four CPU cores. In all mixes, the GPU workload co-executes with the CPU application(s) drawn from the SPEC CPU 2006 suite. All SPEC CPU 2006 applications use the ref inputs. A mix that exercises n CPU cores and the GPU will be referred to as an nC1G mix. Figure 1.1 shows, for each mix (identified by the constituent CPU workload on the x-axis), the performance of CPU and GPU workloads separately relative to the standalone performance of these workloads. For example, for the first 4C1G mix using four CPU cores and the GPU, the standalone CPU performance is the average turn-around time of the four

CPU applications started together, while the GPU is idle. Similarly, the standalone GPU performance is the time taken to complete the Parboil LBM application on the integrated GPU. When workloads on both CPU and GPU run together, performance degrades. As Figure 1.1 shows, the loss in CPU performance varies from 8% (410.bwaves in the 1C1G group) to 28% (the first mix in 4C1G group). The GPU performance degradation ranges from 1% (the first mix in 1C1G group) to 26% (the first mix in 4C1G group). Moreover, the GPU workload degrades more with the increasing number of active CPU cores. These results clearly indicate that without a careful management of shared resources performance degradation in both CPU and GPU workloads can be significant in such processors. When the GPU executes 3D scene rendering workloads, this performance degradation can severely impact the quality of the end-user's visual experience.

1.1 Dissertation Objective and Summary

In this dissertation, we explore solutions to reclaim some part of the lost performance due to CPU-GPU resource contention. Our target heterogeneous chip-multiprocessor architecture has a set of dynamically scheduled out-of-order issue x86 cores integrated with a GPU capable of running 3D scene rendering workloads as well as general-purpose computing applications. The CPU cores and the GPU share the on-die interconnect, the last-level cache (LLC), the memory controllers, and the DRAM banks. We execute memory sensitive applications on the CPU cores, while the GPU is kept busy with either 3D scene rendering applications or general-purpose computing applications.

The traditional CPU core pipeline needs to access the memory hierarchy for

fetching instructions and data. A typical graphics processing pipeline accesses the memory hierarchy for fetching various types of data touched by the fixed function units as well as the programmable shader cores. These include polygon vertices, vertex indices, depth buffer (Z buffer holding pixel depth values), hierarchical depth buffer (HiZ buffer holding hierarchical depth values to reduce Z buffer bandwidth), render targets or render buffers (holding pixel color data), texture maps (input statically as well as generated dynamically), shader instructions, and shader data (including shader register spills and fills). While a 3D scene rendering application can generate accesses to all these different data streams, a general-purpose computing application running on the GPU (usually referred to as GPGPU application) exercises only a portion of the rendering pipeline (primarily the shader cores) and doesn't need to access all these data types. All these GPU streams and the CPU instruction and data streams contend for the shared memory system resources. If these streams are not carefully managed, they interfere with each other leading to significant loss in CPU and GPU performance accompanied by degradation in GPU-rendered 3D animation quality.

To understand the characteristics of the GPU access streams, we start by exploring their memory access behavior (studies are presented in Chapter 2). We find that these streams differ widely in terms of data reuse characteristics and interdependence imposed by the GPU program semantics and idioms, which can be used to improve LLC performance of GPU accesses. Better LLC management of GPU streams can not only improve GPU's performance, but also free up significant amount of DRAM bandwidth consumed by the GPU application. The resources, thus freed, can now be used by the CPU applications. The first contribution of the dissertation explores novel shared LLC management policies for the CPU-GPU

heterogeneous processors.

We evaluate our proposal through simulation studies conducted on a detailed heterogeneous CMP simulator modeling one GPU and four CPU cores. The simulation results show that compared to a state-of-the-art baseline with a 16 MB shared LLC, our proposal can improve the performance (frame rate or execution cycles, as applicable) of eighteen GPU workloads spanning DirectX and OpenGL game titles as well as CUDA applications by 12% on average and up to 51%. The performance of the co-running quad-core CPU workload mixes improves by 7% on average and up to 19%.

3D scene rendering is an important application for the GPU. Since these applications have a clear quality of service (frame rate) requirement, delivering performance above what is required for an adequate visual experience is unnecessary. Based on this observation, we investigate mechanisms that can dynamically shift the memory system resources from GPU to CPU applications in the phases where the GPU application is able to meet the target QoS. We find that the existing resource shifting algorithms target either the LLC capacity [71] or the DRAM bandwidth [46] alone, leading to sub-optimal solutions. In this dissertation, we show that throttling GPU accesses guided by accurate QoS measurements can effectively shift both LLC capacity and DRAM bandwidth from GPU to CPU offering much higher gains than the previous approaches. This study constitutes the second contribution of the dissertation. Detailed simulations done on a heterogeneous CMP model with one GPU and four CPU cores running heterogeneous mixes of DirectX, OpenGL, and CPU applications show that our proposal improves the CPU performance by 18% on average.

The second contribution of the dissertation leaves out a significant fraction of

the heterogeneous workload mixes where the GPU either executes a general-purpose workload (these workloads do not have any well-defined target QoS) or is unable to meet the target QoS of 3D scene rendering. The third part of the dissertation handles the heterogeneous workload mixes containing these GPU applications. To improve the system performance of the heterogeneous mixes containing these GPU applications, we develop mechanisms to dynamically identify the performance-critical GPU access streams and offer a bigger proportion of the DRAM bandwidth to these streams. The goal of such a DRAM bandwidth partitioning scheme is to improve the GPU performance without degrading the CPU performance much. Our GPU access criticality-driven proposal is motivated by the observation that the access streams originating from different GPU units are not equally important for GPU performance. Detailed simulations done on a heterogeneous CMP model with one GPU and four CPU cores running heterogeneous mixes of DirectX, OpenGL, and CPU applications show that our proposal improves the GPU performance by 15% on average without degrading the CPU performance much. The GPGPU applications do not have any minimum QoS requirement. The goal is to improve the performance of these applications as much as possible. We propose extensions to our core mechanisms to handle the heterogeneous workload mixes containing GPGPU applications. These extensions improve the heterogeneous system performance by 7% on average for these mixes, where the heterogeneous system performance is defined as a performance metric that assigns equal weights to CPU and GPU performance.

We summarize the contributions of this thesis in the following.

- We present a mechanism to estimate the dynamic read and write reuse probabilities of LLC blocks and employ these estimates to design novel policies for managing the contents of the LLC shared between the GPU and the CPU

cores.

- We propose a model for dynamic estimation of GPU frame rate and employ these estimates to design a GPU access throttling mechanism for the applications where the GPU already meets the target minimum QoS level. The memory system resources thus freed due to GPU access throttling are dynamically shifted to the CPU cores for improving their performance.

- We propose novel algorithms to dynamically classify the GPU access streams based on their performance-criticality. We employ this classification to design DRAM access scheduling mechanisms that dynamically partition the memory bandwidth between critical GPU accesses, non-critical GPU accesses, and CPU accesses.

The dissertation is organized as follows. Chapter 2 presents a detailed architectural model of the heterogeneous CMP and characterizes the CPU and GPU workloads used in this study. Chapter 3 presents our proposal on shared LLC management policy, which uses dynamic read and write reuse probabilities of LLC blocks to partition the LLC capacity between the CPU and the GPU streams. A model for the GPU frame-rate prediction and the proposal to dynamically throttle the GPU accesses for the applications where the GPU is able to meet the target QoS are presented in Chapter 4. Chapter 5 discusses the algorithm that dynamically classifies the GPU streams based on their criticality toward performance. We also present the proposal for the DRAM scheduling policy to partition the memory bandwidth between the critical GPU accesses, non-critical GPU accesses, and the CPU accesses. Chapter 6 concludes the dissertation.

Chapter 2

Workload Characteristics and Motivation

The heterogeneous system modeled in this dissertation integrates a GPU with multiple CPU cores on the same chip. Figure 2.1 illustrates the full system architecture of the processor. A bidirectional ring connects the CPU cores, the GPU, LLC banks, and two single-channel memory controllers co-located on the chip. The GPU parts modeled include the fixed-function units as well as the programmable shader cores, which are required to execute both the general-purpose applications written using the CUDA APIs and the 3D rendering applications written using the DirectX/OpenGL APIs.

2.1 System Configuration and Target Workloads

The CPU part of the heterogeneous CMP models a dynamically scheduled out-of-order issue x86 core clocked at 4 GHz frequency using the Multi2sim simulation

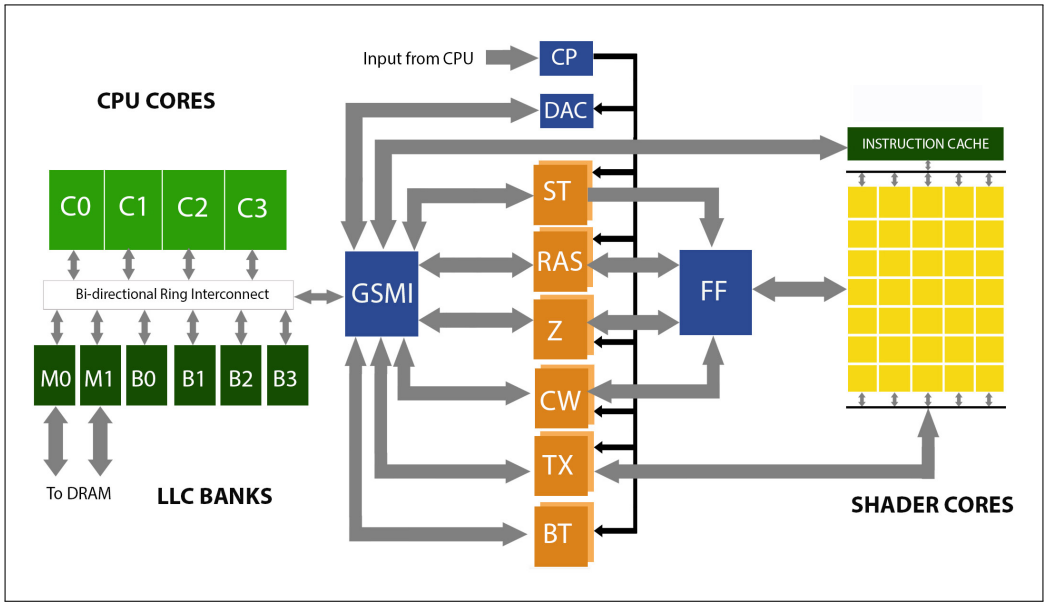


Figure 2.1: CPU-GPU pipeline

infrastructure [102]. Each core has private L1 and L2 caches. The L1 instruction and data caches are 32 KB in size and eight-way set-associative. The unified L2 cache is 256 KB in size and eight-way set-associative. The L1 and L2 cache lookup latencies are two and three cycles, respectively (determined using CACTI [27] for 22 nm node). The L1 and L2 caches have a block size of 64 bytes.

We do not use the GPU model that comes with Multi2Sim. Instead, we use two GPU simulators, one to execute the 3D scene rendering jobs and the other to execute the CUDA applications. The 3D scene rendering GPU is modeled with an upgraded version of the Attila GPU simulator [73]. The simulator has enough details to capture all the phases of the entire rendering pipeline. The simulated GPU uses a unified shader model where the same set of shader cores is used to carry out vertex shading as well as pixel (or fragment) shading. The GPU has 64 shader cores clocked at 1 GHz. Each shader core has four ALUs and each ALU is equipped with a 4-way SIMD vector unit and a scalar unit. Thus, each shader core has a peak throughput of

16 single-precision floating-point operations every cycle leading to an overall single-precision floating-point throughput of one tera-FLOPS for the GPU. The GPU has enough register resources to maintain 4096 in-flight shader thread contexts, where each thread, when scheduled on a shader core, can issue four 4-way SIMD operations in a cycle.[1] The shader core scheduler executes a round-robin scheduling algorithm among the ready thread contexts. A running thread changes state to blocked when it issues either a branch instruction or a texture load instruction. Each shader core is attached to two texture samplers. Each texture sampler can process one 32-bit texel every cycle giving rise to a peak texture fill rate of 128 GTexels/second. The simulated GPU has sixteen render output pipeline (ROP) units. The ROP units receive quad-pixel stamps after they are processed by shader cores. Each ROP has a depth test unit, a pixel color blending unit, and a color writer unit that writes out the final pixel color to the render target. Each of these units can process one quad-pixel stamp every cycle leading to a peak pixel fill rate of 64 GPixel/second.

The GPU has a three-level non-inclusive texture cache hierarchy resembling the texture cache hierarchy of Intel's integrated GPUs (Gen7 onward). The L0 texture cache is 2 KB fully-associative and private to each sampler. The L1 texture cache is 64 KB 16-way set-associative and shared by all the samplers. The L2 texture cache is 384 KB 48-way set-associative and shared by all the samplers. All texture caches have a block size of 64 bytes. Each ROP unit is equipped with a 2 KB fully-associative L1 depth cache and a 2 KB fully-associative L1 color cache with block size of 256 bytes. The non-inclusive L2 depth and color caches are each 32 KB 32-way set-associative with 64-byte blocks and shared by all ROP units. Additionally, the simulated GPU has a fully-associative 16 KB vertex cache, a 16 KB 16-way

[1] A thread context is equivalent to a 16-way warp in Nvidia terminology.

hierarchical depth (HiZ) cache, and a 32 KB 8-way shader instruction cache.

The GPU model used for executing the CUDA applications is borrowed from the MacSim simulator [55], which makes use of the GPUOcelot [16] tool for capturing the CUDA application instructions. Since the CUDA applications make use of the shader core only, the GPU simulator contains a detailed model of the shader core island of the GPU. In this configuration, the GPU has sixteen shader cores, each clocked at 2 GHz and each having resources to maintain a maximum of eighty warp contexts (each warp has 32 threads). The instruction scheduler of each shader core selects two ready warps from the pool of eighty ready warps every four cycles. The peak execution throughput of each shader core is sixteen single-precision floating point operations per cycle leading to 512 GFLOPS for the entire GPU. Each shader core is equipped with a 4 KB eight-way instruction cache, a 32 KB eight-way data cache, an 8 KB texture cache, an 8 KB constant cache, and a 16 KB software-managed shared memory.

The CPU and GPU simulators are integrated through a graphics to system memory interface (GSMI) module. The cache subsystem of the GPU simulators is enhanced so that it can access GSMI for shared memory accesses. Figure 2.2 depicts a high-level view of the CPU-GPU integration. Depending on the type of the GPU workload being executed (workload type is provided as a simulation parameter), one of the two GPU models is instantiated and attached to the rest of the heterogeneous CMP through the GSMI object. Since the instantiated GPU simulator has access to the GSMI object, it can issue requests to the shared memory if required. The shared LLC of the heterogeneous CMP receives requests that miss in the CPU cores' L2 caches or GPU's private caches. The LLC is 16 MB sixteen-way set-associative with a lookup latency of ten cycles. The LLC maintains inclusion for all CPU instruction

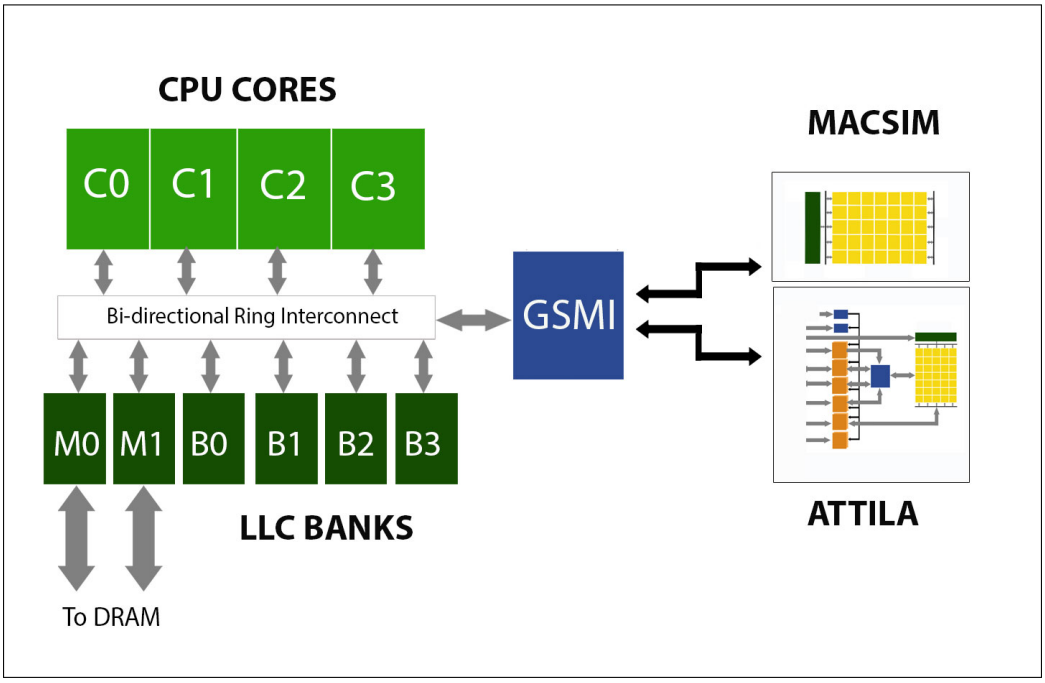


Figure 2.2: Block diagram of the heterogeneous simulator integrating Multi2Sim, Attila, and Macsim

and data. However, the GPU data are not kept inclusive in the sense that on an LLC eviction, a back-invalidation is not sent to the GPU's internal caches. Such a design decision also keeps open the option of bypassing the LLC on an LLC read miss for GPU data. The current model does not support coherence between GPU and CPU address spaces. As a result, all evaluations done in this thesis are limited to heterogeneous workloads that do not exercise any application-level sharing between the GPU and CPU data. The GPU accesses the LLC through GSMI for all private cache misses. GSMI contains a set of request queues and a cross-bar to route requests from different units to the ring interconnect.

Figure 2.3 shows a detailed schematic of the GSMI. The GPU connects to the GSMI through the input and response queues. The crossbar connects input queues to the request queue through a Graphics Translation Table (GTT). Every request

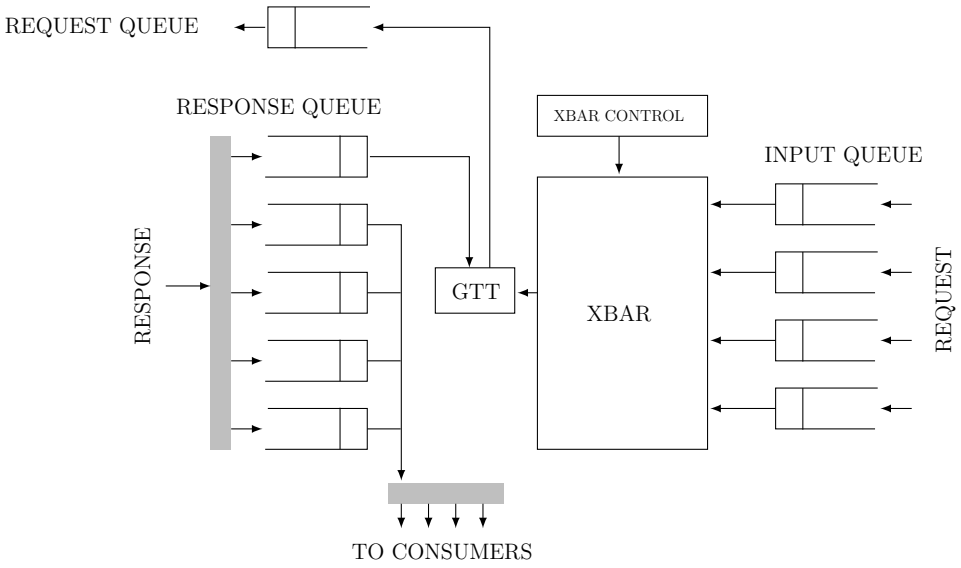


Figure 2.3: Graphics to System Memory Interface

that is sent to the LLC from the GSMI goes through GPU to physical address translation. A memory request is submitted to GSMI by enqueuing it to the input queue. Every cycle for a set of requests currently in the input queue, the GTT lookup is carried out and address translation is performed. Since the page table entries (PTEs) are cached in the LLC, on a GTT miss, the PTE is fetched from the LLC if it is found there. The PTE fetch follows the same request path as is followed by all other GPU memory requests. Once the translation succeeds, the request is enqueued in the request queue. On the return path, a response is enqueued into a per-unit response queue. These responses are finally consumed by the respective GPU units. Our model supports 4 GB GPU address space and one level of GTT. The GTT is four-way ported and has 32K entries. To feed the GTT, the crossbar has four output ports and the request queue accepts four inputs per cycle. We also support render target and texture surface tiling. The address presented to GSMI is already translated from tiled to linear address space.

The simulated heterogeneous CMP is equipped with two on-die single-channel memory controllers. Each memory controller connects to a 2 GB DRAM module modeled using DRAMSim2 [89]. Each DRAM module is eight-way banked single-rank DDR3-2133 with 14-14-14 (CL-RCD-RP) latency parameters and burst length of eight. The memory controllers implement the FR-FCFS scheduling algorithm. The CPU cores along with their private caches, the GPU, the LLC, and the memory controllers are arranged on a bidirectional ring interconnect having a single-cycle hop time. The choice of a ring interconnect is motivated by its simplicity and low area overhead. A ring interconnect can comfortably support a moderate number of communicating agents, which is the case for our model having four CPU cores with the LLC banks, one GPU, and two DDR3-2133 memory controllers. Additionally, a bidirectional ring has a reasonably low average hop count when the number of agents on the ring is low. Recent client processor offerings with integrated GPU from Intel have similar microarchitecture and employ a ring interconnect [42, 43, 93]. Table 2.1 summarizes the salient parameters of the hardware model.

The heterogeneous workloads are built by mixing CPU applications drawn from the SPEC 2006 suit and 3D scene rendering jobs drawn from fourteen popular DirectX and OpenGL game titles as well as six CUDA applications from publicly available benchmark suits. The DirectX and OpenGL API traces for the selected 3D animation frames are obtained from the Attila simulation distribution [73] and 3DMark06 suit [110]. The simulated game regions (i.e., sequence of frames) are selected at random after skipping over the initial sequence and detailed in Table 2.2. The last column of this table lists the average frame rate of each game region in frames per second (FPS) when the rendering job is co-scheduled with a mix of four CPU applicarions in a 4C1G configuration. The details of selected CUDA applica-

tions are shown in Table 2.3. The graphics API traces or the CUDA application's shader instruction traces (as applicable) are replayed through the GPU simulator, while the selected mixes of the SPEC CPU 2006 applications are simulated in execution-driven mode on the CPU cores. Table 2.4 lists the 1C1G, 2C1G, and 4C1G workload mixes (S1-S18, D1-D18, Q1-Q18) used in this study.

The GPU workload is same in Sn, Dn, and Qn for a given n. One, two, and four different memory-sensitive SPEC 2006 applications are drawn at random and associated with the GPU workload to complete the mix Sn, Dn, and Qn, respectively. Each CPU application in a mix commits at least 250 million representative dynamic instructions and an early-finishing application continues to run until each CPU application commits its representative set of dynamic instructions and the GPU completes rendering the set of 3D frames or the portion of the CUDA application assigned to it. The performance for the CPU mixes is measured in terms of average instruction per cycle throughput. The GPU performance for the 3D scene rendering jobs is measured in terms of average frame rate and for CUDA applications, the number of execution cycles to complete the jobs is used.

2.2 Execution Model of 3D Rendering

The execution models of compute pipeline (used in the CPU cores and the GPU shader cores) and 3D rendering pipeline differ vastly. Figures 2.4 and 2.5 show the stages of typical compute and 3D rendering pipelines, respectively. A compute pipeline executes instructions accessing memory only in two stages for instructions and data. On the other hand, a 3D rendering pipeline accesses various buffers allocated in memory for storing rendering state and data throughout the

Table 2.1: Simulation environment

CPU cache hierarchy	
Per-core iL1 and dL1 caches	32 KB, 8-way, 2 cycles
Per-core unified L2 cache	256 KB, 8-way, 3 cycles
GPU model for 3D scene rendering	
Shader cores	64, 1 GHz, four 4-way SIMD per core
Texture samplers	two per shader core, 128 GTexel/s
ROP:	16, fill rate 64 GPixels/s
Texture caches	three-level hierarchy, L0: 2 KB per sampler, shared L1, L2: 64 KB, 384 KB
Depth caches	two-level hierarchy, L1: 2 KB per ROP, shared L2: 32 KB
Color caches	two-level hierarchy, L1: 2 KB per ROP, shared L2: 32 KB
Vertex cache	16 KB, shader instruction cache: 32 KB,
Hierarchical depth cache:	16 KB
GPU model for general-purpose computing	
Shader cores	16, 2 GHz, sixteen SP FLOPs/cycle
Per-core instruction, data cache	4 KB, 32 KB,
Per-core texture, constant cache:	8 KB, 8 KB,
Per-core shared memory:	16 KB
Shared LLC and interconnect	
Shared LLC	16 MB, 16-way, lookup latency 10 cycles, inclusive for CPU blocks, non-inclusive for GPU blocks, two-bit SRRIP policy [35]
Interconnect	bidirectional ring, single-cycle hop
Memory controllers and DRAM	
Memory controllers	two on-die single-channel, DDR3-2133, FR-FCFS access scheduling in baseline
DRAM modules	14-14-14, 64-bit channels, BL=8, open-page policy, one rank/channel, 8 banks/rank, 1 KB row/bank/device, x8 devices

Table 2.2: Graphics frame details

Application	DirectX/ OpenGL	Frames	Resolution	FPS
3DMark06 GT1	DirectX	670–671	1280×1024	5.9
3DMark06 GT2	DirectX	500–501	1280×1024	14.0
3DMark06 HDR1	DirectX	600–601	1280×1024	16.7
3DMark06 HDR2	DirectX	550–551	1280×1024	21.8
Call of Duty 2 (COD2)	DirectX	208–209	1920×1200	19.5
Crysis	DirectX	400–401	1920×1200	6.7
DOOM3	OpenGL	300–304	1600×1200	80.7
Half Life 2 (HL2)	DirectX	25–27	1600×1200	77.4
Left for Dead (L4D)	DirectX	601–605	1280×1024	33.6
Need for Speed (NFS)	DirectX	10–12	1280×1024	66.6
Quake4	OpenGL	300–304	1600×1200	80.5
Chronicles of Riddick (COR)	OpenGL	253–257	1280×1024	103.9
Unreal Tournament 2004 (UT2004)	OpenGL	200–204	1600×1200	132.5
Unreal Tournament 3 (UT3)	DirectX	955–957	1280×1024	26.6

Table 2.3: CUDA application details

Application	Source	Thread configuration
LBM	CUDA SDK 4.2	120×150 blocks, 120 threads/block
CFD	Rodinia 3.0	759 blocks, 128 threads/block
BFS	Rodinia 3.0	1954 blocks, 512 threads/block
FASTWALSH	CUDA SDK 4.2	8192 blocks, 256 threads/block
BLACKSCHOLES	CUDA SDK 4.2	480 blocks, 128 threads/block
REDUCTION	CUDA SDK 4.2	64 blocks, 256 threads/block

pipeline. Moreover, a GPU exercises various fixed-function units in addition to the programmable shader cores while rendering a 3D scene. These units are (shown in Figure 2.1) command processor (CP), display controller (DAC), vertex streamer (ST), rasterizer (RAS), depth and stencil test (Z), color blender and writer (CW), texture sampler (TX), and blitter (BT). The programmable shader cores use a uni-

Table 2.4: Heterogeneous workload mixes

GPU workload	CPU workload mix		
	1C1G	2C1G	4C1G
3DMark06 GT1	S1: wrf	D1: mcf milc	Q1: gcc.166.i, soplex.pds-50, sphinx3, wrf
3DMark06 GT2	S2: omnetpp	D2: bwaves milc	Q2: gcc.166.i, mcf, sphinx3, zeusmp
3DMark06 HDR1	S3: lbm	D3: bzip2.source lbm	Q3: bzip2.source, lbm, leslie3d, soplex.pds-50
3DMark06 HDR2	S4: sphinx3	D4: lbm libquantum	Q4: bzip2.source, lbm, libquantum, omnetpp
COD2	S5: lbm	D5: bzip2.source lbm	Q5: bzip2.source, lbm, leslie3d, soplex.pds-50
Crysis	S6: mcf	D6: soplex.pds-50 wrf	Q6: mcf, milc, sphinx3, zeusmp
DOOM3	S7: libquantum	D7: libquantum omnetpp	Q7: bwaves, libquantum, milc, omnetpp
HL2	S8: gcc.166.i	D8: bwaves omnetpp	Q8: bwaves, mcf, milc, zeusmp
L4D	S9: libquantum	D9: libquantum omnetpp	Q9: bwaves, libquantum, milc, omnetpp
NFS	S10: leslie3D	D10: gcc.166.i wrf	Q10: bwaves, mcf, milc, omnetpp
Quake4	S11: bwaves	D11: mcf zeusmp	Q11: bzip2.source, leslie3d, soplex.pds-50, wrf
COR	S12: zeusmp	D12: bzip2.source leslie3D	Q12: gcc.166.i, leslie3d, soplex.pds-50, wrf
UT2004	S13: soplex.pds-50	D13: sphinx3 zeusmp	Q13: bzip2.source, lbm, leslie3d, libquantum
UT3	S14: zeusmp	D14: bzip2.source leslie3D	Q14: gcc.166.i, leslie3d, soplex.pds-50, wrf
cfd	S15: sphinx3	D15: lbm libquantum	Q15: bzip2.source, lbm, libquantum, omnetpp
blackscholes	S16: gcc.166.i	D16: libquantum omnetpp	Q16: bwaves, libquantum, milc, omnetpp
fastwalsh	S17: mcf	D17: libquantum omnetpp	Q17: bwaves, libquantum, milc, omnetpp
reduction	S18: libquantum	D18: gcc.166.i wrf	Q18: bwaves, mcf, milc, omnetpp

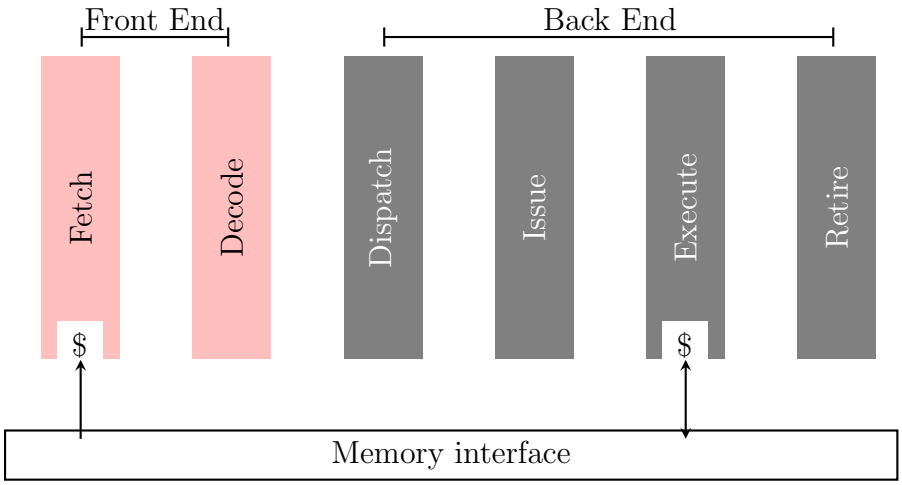


Figure 2.4: A typical compute pipeline

fied shader execution model, where the same set of cores execute both the vertex and fragment shader programs. All GPU units coordinate with each other through the fragment FIFO (FF) unit, which contains a crossbar and a scheduler to steer inputs from source to destination units.

CPU cores access the LLC on an L2 cache miss. For GPU, on the other hand, the private cache hierarchy differs across units. Memory requests from DAC and BT always go to the LLC, since they do not have any private caches. For all other units, requests pass through one or more levels of private caches before accessing the LLC. ST reads vertex attributes from a vertex buffer before sending it to the Shader. To speedup the fetch, it caches vertex data in a private vertex cache backed by the LLC. All instances of Z and CW units use a multi-level cache hierarchy for the render target and depth buffer. The inner level in these units is private to each instance, while the outer level is shared by all of them. The accesses missing in the shared level are sent to the LLC. The shader cores fetch instructions through a single level shared instruction cache. TX accesses the texture map data through

a multi-level cache hierarchy. The first level of this hierarchy is private to the each sampler instance, while the next two levels are shared by all instances. RAS uses an LLC-backed hierarchical depth (HiZ) cache for caching depth values to carry out hierarchical depth tests. A comprehensive list of cache configuration for various GPU units is presented in Table 2.1.

2.2.1 Overview of the 3D Rendering Algorithm

The 3D rendering algorithm, implemented in our GPU model, first assembles shaded vertices into geometric primitives (i.e., triangle), and subsequently breaks them into fragment quads. Each fragment contains all the information needed to generate the pixels for a given fragment. These fragment quads go through various tests (occlusion, alpha testing, etc.) and shading operations to decide their visibility and the final attribute values. Ultimately, the computed color values are blended and written back to the render target buffer.

A 3D scene to be rendered is presented to the GPU as batches of geometric primitives by an application. To better utilize the resources, a GPU pipeline is divided into two sub parts, namely, front- and back-end (shown in Figure 2.5). This division enables processing of two different batches simultaneously in two sub parts. We define a few coarse-grain states of the GPU in the following. The GPU is said to be in the Draw (DW) state, when both front and back-end are active. It is said to be in the EndGeometry (EG) state, when the front-end is idle, but the back-end is active. Similarly, if the back-end is idle but the front-end is active, it is said to be in the EndFragment (EF) state. These states reflect the pipeline utilization at a coarse granularity. For example, in the DW state, the GPU is fully utilized because both

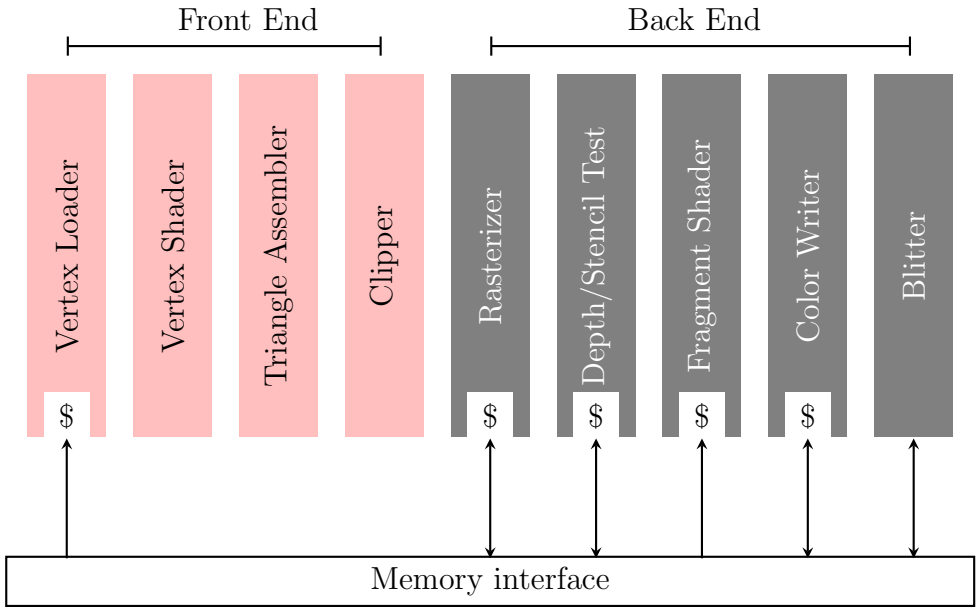


Figure 2.5: Stages of a 3D rendering pipeline

the front- and back-end are active. However, in the EG or EF state, only back- or front-end is active and hence, the pipeline is partially utilized. In addition to these three states, the GPU can be in the Blit (BT) or the SaveRestore (SR) state. In the BT state, only the blitter part of the GPU remains active. The blitter is used to perform a fast copy of a dynamically generated texture surface from the render target buffer. In the SR state, the internal states (used for compressed color/depth buffer and fast clear operations) maintained for the render target and depth buffer are saved and restored from the system memory. During this time, the rest of the GPU pipeline remains inactive. The GPU controls the activity of different units using a state machine shown in Figure 2.6. As the diagram shows, in the beginning of a frame, when all GPU units are idle, the GPU is said to be in the Ready (RDY) state. In this state, the GPU can accept any command or state updates from the CPU. From the RDY state, the GPU can transition to one of three different states

namely, DW, BT, and SR. A Draw command from the CPU transitions it to the DW state. Similarly, a Blit or SaveRestore command changes the GPU's state to BT or SR, respectively. As soon as the SaveRestore or the Blitting operation finishes, the GPU again returns to the RDY state. From the DW state, the GPU moves to the EG or EF state based on the situation in the front- and back-end units. If the front-end has finished processing of the batch, the GPU moves to the EG state. Similarly, if the back-end has finished, the GPU moves to EF state. If both the front- and back-end have finished, the GPU goes back to the RDY state.

The front-end of the GPU pipeline consists of four stages, namely vertex streamer, vertex shader, triangle assembler, and clipper. The role of the vertex streamer is to load vertex indices from the index buffer, and if necessary, vertex attributes from the vertex buffer. Once these attributes are available, the vertex shader stage runs a shading program on the loaded data values. To eliminate some amount of redundancy, for two primitives that share a vertex, the same shaded vertex attributes are passed to the triangle assembler stage that connects and builds a primitive from the individual vertices. In the final stage, the clipper carries out the view frustum clipping of the assembled primitives. Apart from the vertex streamer, all other stages of the front-end are compute intensive in nature. Memory access is needed only to fetch the vertex indices and attributes.

Most of the memory-intensive operations take place in the back-end of the pipeline. This part of the pipeline consists of five stages, namely rasterizer, depth and stencil test, fragment shader, color blender and writer, and blitter. Each of these stages needs to access memory to load attribute values for performing a test or to calculate the new values. The rasterizer traverses the input polygons and generates tiled fragments corresponding to each pixel. Tiling of fragments improves the

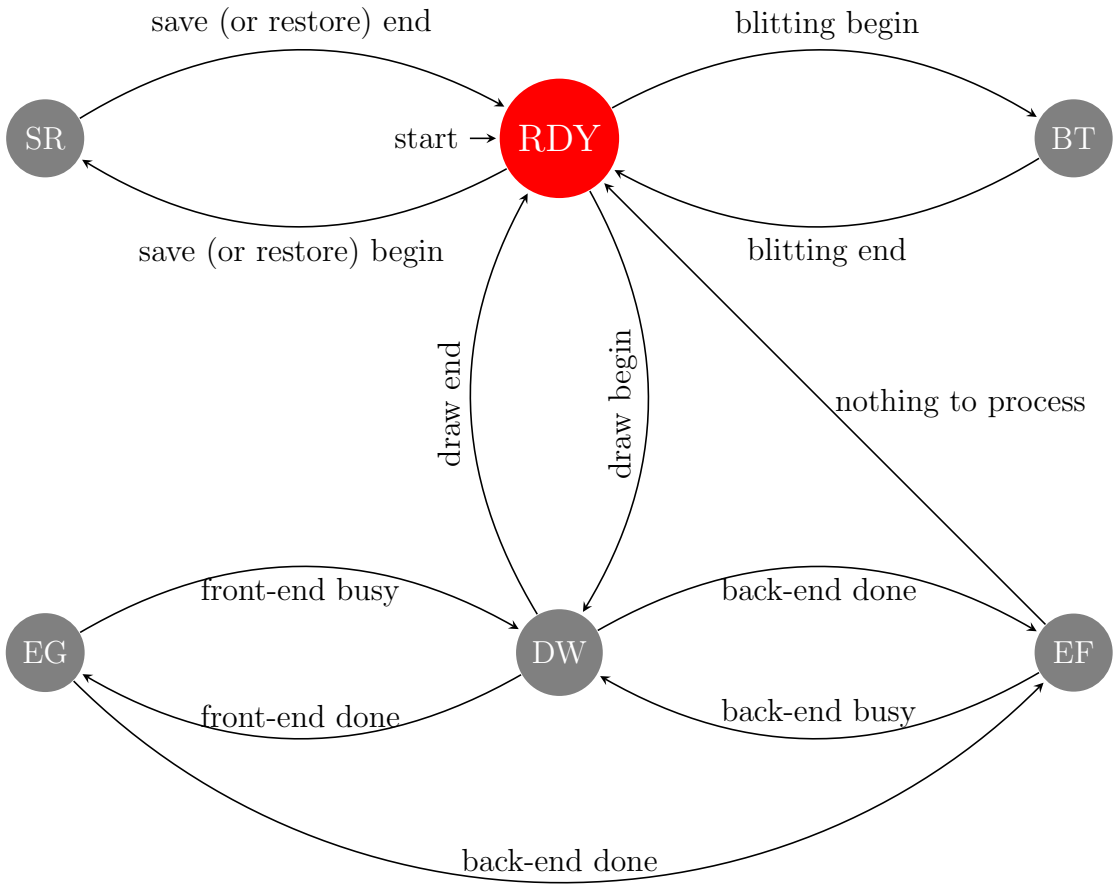


Figure 2.6: GPU state machine

locality of memory accesses in the subsequent pipeline stages. The generated fragments contain all the attributes required to obtain the final pixel color. Depth and stencil test and fragment shading can happen in two different orders. If fragment shading happens after the depth and stencil tests, it is called early-Z order. If it happens before the depth and stencil tests, the order is called late-Z. Whether the pipeline will operate in early-Z or late-Z order is decided by the GPU driver. For this purpose, the driver analyzes the operations in the fragment shader program,

and if the program modifies the depth values, the pipeline operates in the late-Z order; otherwise it operates in the early-Z order. After shading and depth/stencil test, the fragments that survive depth test proceed to the color blending and write stage. In this stage, based on the current blending state, the final fragment colors are computed and written to the render target. For the operations, such as, changing resolution of a scene or generating texture map from the render target for dynamic texturing, the ability to quickly copy the current render target to a texture map is required. These texture maps are subsequently sampled and reconstructed to get the desired scene. Blitter, a special purpose unit, performs such high-speed memory to memory copy. Based on the render target and texture surface formats, the blitter also performs compression/decompression and format translation before writing the pixel values to the destination texture map. Once the processing of the fragments is finished, the DAC reads the color values from the memory and sends them to the display port.

2.3 Performance Impact of Resource Sharing

Ideally, a heterogeneous CMP should make simultaneous use of both CPU and GPU cores to maximize the utilization of on-chip resources. As the CPU and the GPU working sets contend for the shared LLC capacity and the shared DRAM resources, the destructive interference arising from this contention can significantly hamper the progress of the tasks being executed by the CPU and the GPU cores. Most importantly, if the integrated GPU is being used to render 3D scenes, the end-user's visual experience can suffer from unacceptable degradation. To quantify this loss in performance when both types of cores are active, we conduct an experiment where

we run GPU jobs standalone by keeping the CPU core(s) free. CPU jobs standalone by keeping the GPU free, and finally, both types of jobs simultaneously in the heterogeneous mode of execution. Figure 2.7 shows the performance of heterogeneous execution over standalone execution when the simulated CMP is equipped with a shared 16 MB LLC. The CPU and GPU configurations used in this experiment were presented in Table 2.1. The rendering and the CUDA workloads executed on the GPU were presented in Tables 2.2 and 2.3, respectively. The heterogeneous workloads are denoted by S1-S18, D1-D18, and Q1-Q18, respectively for one CPU and one GPU (1C1G), two CPU and one GPU (2C1G), and four CPU and one GPU (4C1G) configurations. The details of these mixes were given in Table 2.4. In Figure 2.7, the top panel shows the results for the 1C1G configuration, the middle panel shows the results for the 2C1G configuration, and the bottom panel shows the results for the 4C1G configuration. The CPU (GPU) bar shows the performance achieved by the CPU (GPU) job when it runs together with a GPU (CPU) job compared to when it runs standalone.

From these results, we see that for a 1C1G CMP, compared to the case when only the CPU is active, the CPU performance degradation is 26% (see the CPU bar in the GMEAN group of the top panel) in the heterogeneous mode. Similarly, the GPU job experiences an average 17% degradation compared to its standalone execution. As the number of active CPU cores increases, the interference experienced by the GPU workloads increases sharply. For a 2C1G CMP, in the heterogeneous mode the dual-core CPU workload degrades by 26% on average compared to the standalone execution, while the GPU degrades by 24% compared to the standalone GPU execution. For a 4C1G CMP, degradation experienced by the heterogeneous CPU and GPU jobs is 21% and 35%, respectively. As expected, with increasing

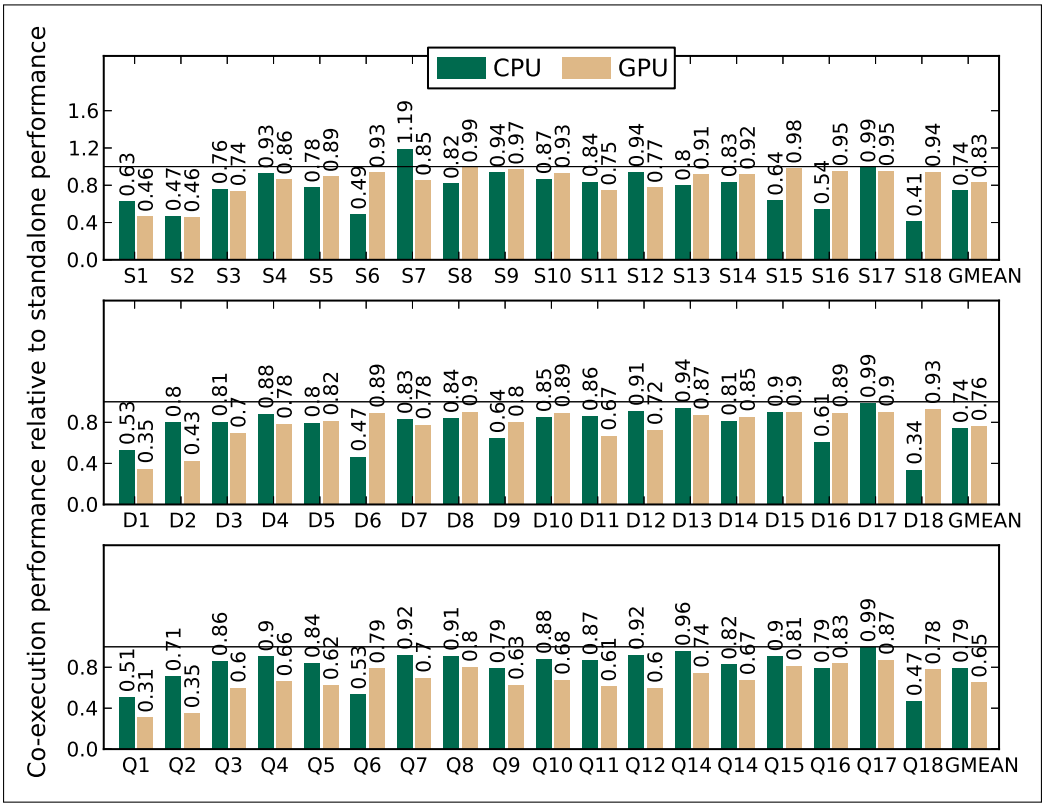


Figure 2.7: Co-execution performance relative to standalone performance with a 16 MB shared LLC

CPU core count, the GPU workloads suffer more compared to the CPU workloads in the heterogeneous mode of execution.

Since for the CPU cores, the only possible source of a memory access is instruction or data fetch, the performance degrades due to the delayed responses to these accesses. However, for the GPU, the memory accesses originate from various fixed function and programmable units. Thus, to pinpoint the sources of degradation for the GPU, we analyze the activity and memory access behavior of different parts of the 3D rendering pipeline. Sections 2.3.1 and 2.3.2 present the utilization and stall contribution of different parts of GPU pipeline, respectively. Section 2.3.3 presents the memory access characteristics of the GPU access streams. These sections use

the 4C1G configuration and the corresponding heterogeneous workload mixes (Qn's) for collecting the analysis data. We focus on only those mixes that have 3D scene rendering workloads as the GPU applications (Q1 to Q14; please refer to Table 2.4). This allows us to understand the dynamics of the entire rendering pipeline, which wouldn't be possible if we study the GPGPU workloads because the GPGPU workloads exercise primarily the shader cores of the rendering pipeline.

2.3.1 GPU Utilization in 3D Rendering

Figure 2.8 shows the percent of execution cycles the GPU spends in various states for each 3D rendering workload. Two major contributors are the Draw and EndGeometry states with an average contribution of 23% and 66% cycles, respectively. The average contributions of Blit and SaveRestore states are 5% and 4% cycles, respectively. During the remaining 2% cycles, the GPU executes in the EndFragment state. Since the back-end remains active in both Draw and EndGeometry states, on average, its utilization is 89%. On the other hand, the front-end remains active for 23% of the cycles on average (the front-end and back-end simultaneously execute during these 23% of the cycles). Although the average contribution of the Blit state is only 5%, there are applications contributing much higher. For example, the Blit contributions in L4D, COR, and COD2 are 29%, 24%, and 16%, respectively, which is much higher than the average. Similarly, the SaveRestore state contributes 23% and 17% for L4D and NFS, respectively. In conclusion, the back-end units remain active most of the time, and the contributions from the other units (the front-end units, the blitter, etc.) are also significant and require further investigation.

The impact of the memory accesses originating from the front-end unit (vertex

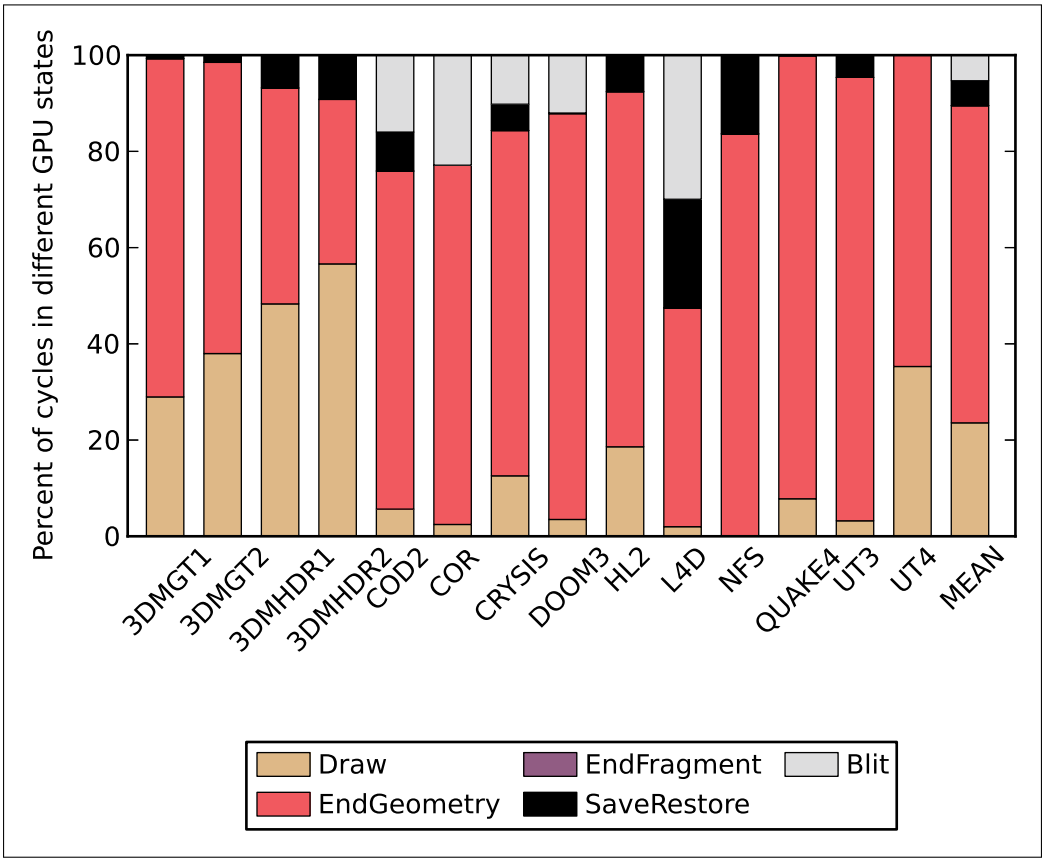


Figure 2.8: Cycles spent in different GPU states. 3DMGT1, 3DMGT2, 3DMHDR1, and 3DMHDR2 are short names for 3DMark06 GT1, 3DMark06 GT2, 3DMark06 HDR1, and 3DMark06 HDR2, respectively.

streamer) and the blitter is confined to that particular unit only. Therefore, the sources of stalls resulting in performance degradation of these units are easily understood. For the back-end, the situation is different, however. Figure 2.9 shows the stall cycles contributed by different back-end units of the GPU pipeline.

2.3.2 Stall Contribution of Different GPU Units

The impact of the memory accesses originating from the front-end unit (vertex streamer) and the blitter is confined to that particular unit only. Therefore, the sources of stalls resulting in performance degradation of these units are easily un-

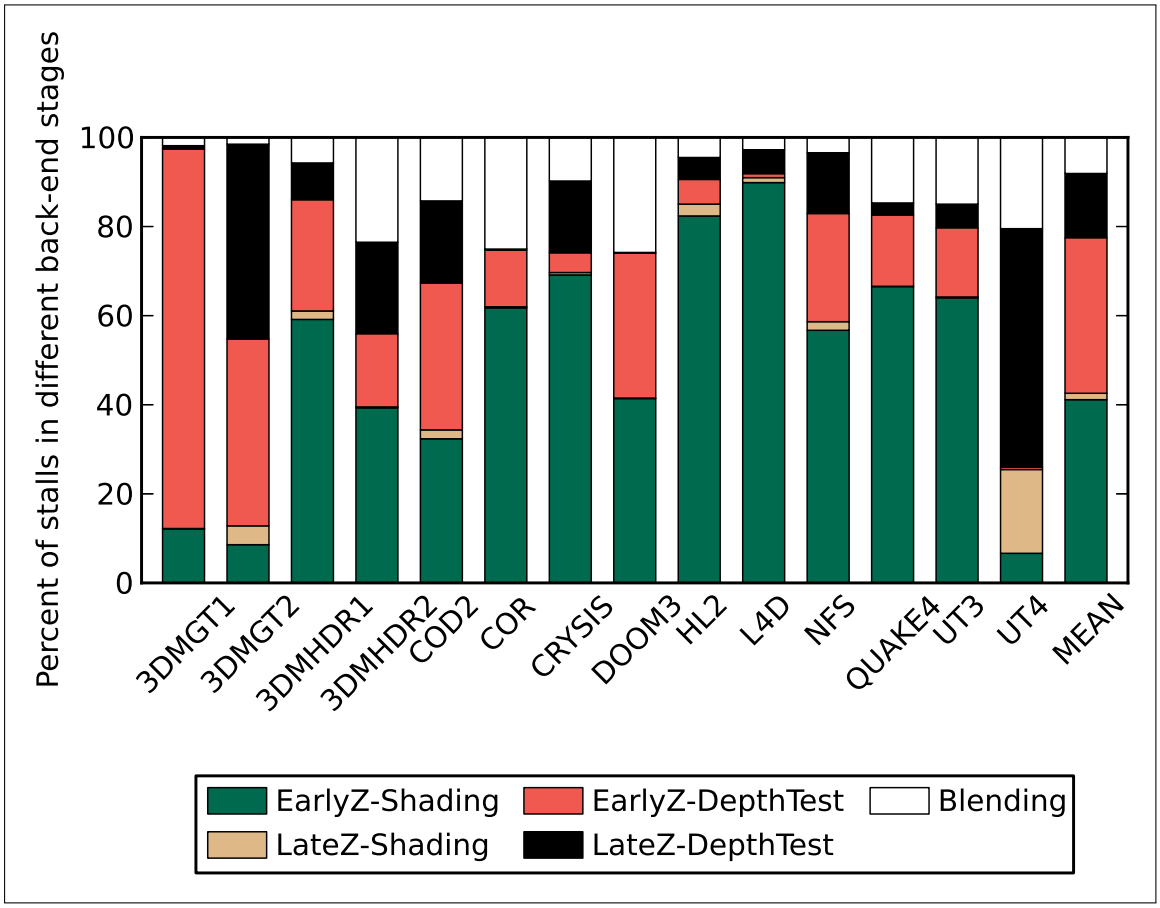


Figure 2.9: Stall cycle distribution in the back-end of the rendering pipeline.

derstood. For the back-end, the situation is different, however. Figure 2.9 shows the stall cycles contributed by different back-end units of the GPU pipeline. On average, fragment shading in the early-Z mode contributes 39% stalls, while its contribution in late-Z mode is less than 5%. The depth test contributes 36% and 14% of stalls, on average, in the early-Z and late-Z modes, respectively. The average contribution of blending is 8%. Importantly, the stall distribution of the streams across the applications is not uniform. For example, the early-Z test contributes 86% stalls in 3DMGT1, while the combined (late-Z and early-Z) depth test in L4D contributes less than 10% stalls. Similar varying impacts across applications are seen for the

other streams as well.

2.3.3 GPU Memory Access Characteristics

We observe that the number of stall cycles do not always correlate directly with the memory access intensity. Figure 2.10 presents the distribution of LLC accesses across three important GPU streams, namely color (C), texture (T), depth (Z), blitter(B), and vertex (I). On average, color, texture, and depth streams contribute

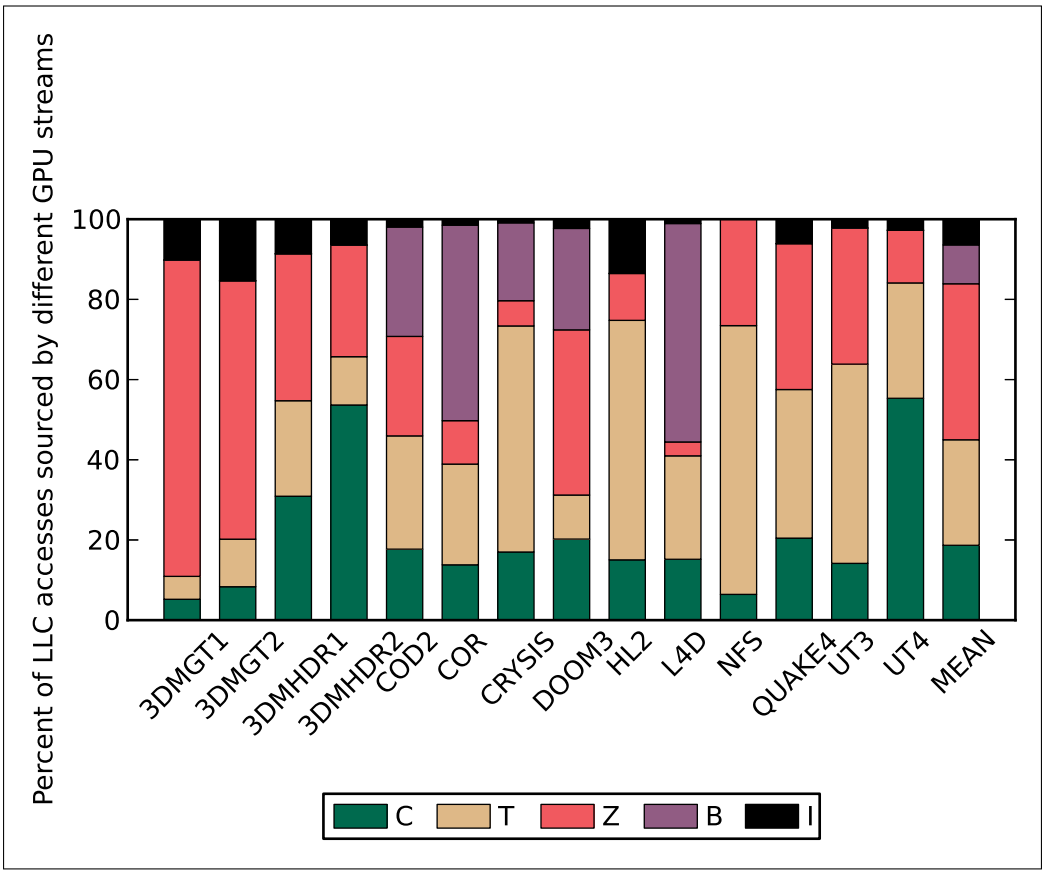


Figure 2.10: LLC access distribution for different units

18%, 26%, and 38% accesses, respectively. Although for the applications such as, 3DMGT1 and 3DMGT2, where the majority of the accesses arrive from one particular stream, stall cycles and access counts correlate well. Similar correlation is

not seen for the other applications, however. For example, for 3DMHDR1, the color stream contributes 30% accesses, while contributing just 5% stalls (the Blending stalls in Figure 2.9 arise from the color stream). On the other hand, shading contributes 60% stalls with only 23% LLC accesses in 3DMHDR1. Similarly, for DOOM3, the depth stream contributes 40% LLC accesses, while causing only 35% stalls (combined late-Z and early-Z stalls in Figure 2.9). Similar trends are seen for the other applications too. These data show that the GPU streams differ in terms of their sensitivity toward memory system performance due to either certain inherent dataflow dependencies in the pipeline or their memory access behavior. One possible reason for differing memory access behavior may arise from variable sensitivity of streams toward caching. Figure 2.11 presents the variation in miss rates of five important GPU streams (color (C), depth (Z), texture (T), blit (B), and input (I)) for four different GPU applications (the data for all applications can be found in Appendix 6.1) as the LLC size is increased from 8 KB to 256 MB. In these experiments, we use LRU replacement policy for the LLC.

As Figure 2.11 shows, for most of the streams, the amount of cache required for the entire working set is more than 32 MB. Moreover, the effect of increase in LLC capacity on miss rate varies across streams and applications. Across all applications, the color stream's miss rate drops below 30% only at the capacity equal to or above 8 MB. For the texture stream, the working set is much larger; the miss rate remains above 30% up to 16 MB capacity. Miss rate of the depth stream shows a gradual reduction for COR, COD2, and 3DMGT1 reaching 30% at 4 MB capacity. However, for L4D, it is much larger and it reaches the 30% mark only at 32 MB capacity. Similarly, for the input stream, the miss rate doesn't go below 30% up until 16 MB capacity. Only COR, COD2, and L4D show accesses from the

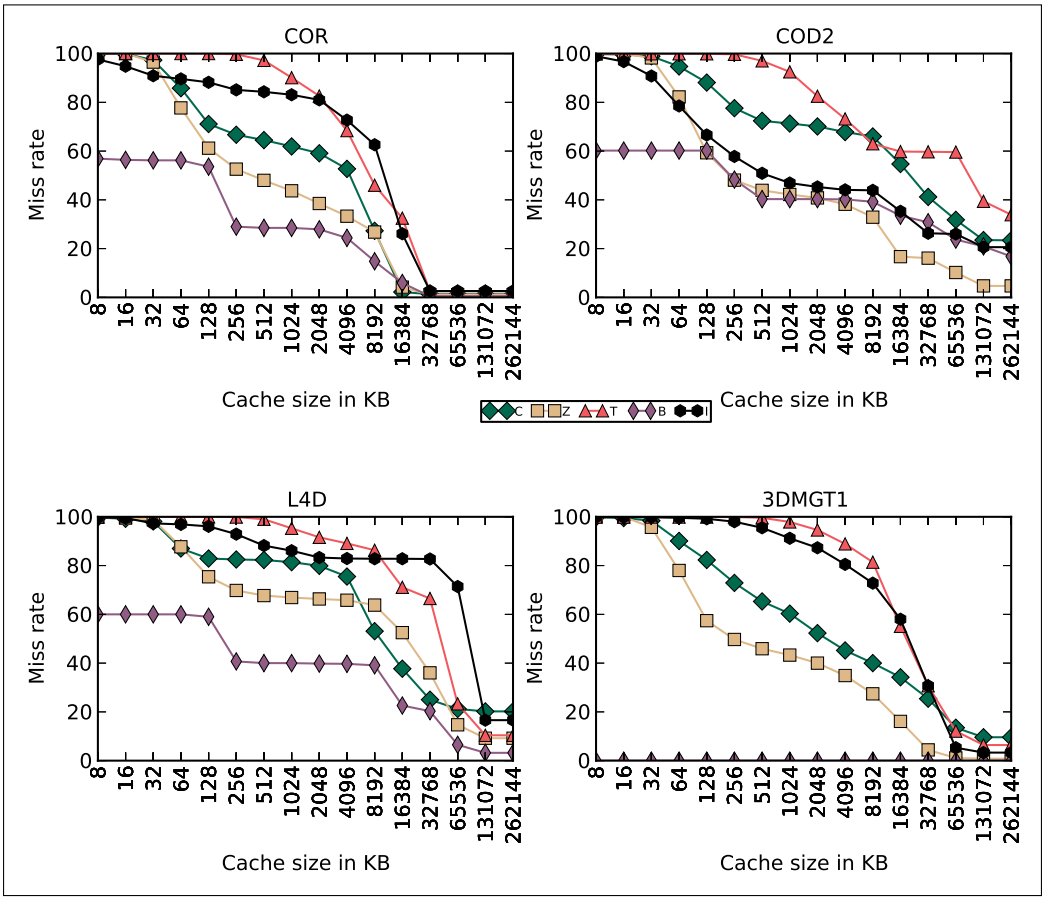


Figure 2.11: Working set of different LLC access streams

blit stream. Interestingly, miss rate for the blit stream never reaches 100%. This behavior is a result of the fact that the blitter always consumes data produced by the color or texture stream (as described in Section 2.2, the blitter is used to copy the blocks from a render target to a texture buffer). Moreover, the reduction in its working set is a step function with a small region of gradual reduction in its miss rate. Similar to C (or T) to B reuse, Z, C, and T streams also exhibit inter-stream reuses due to the semantic requirements of the rendering algorithms. In the next section, we present an analysis of the inter-stream reuses observed between the C, Z, B, and T streams.

2.3.4 Inter-stream Reuses in 3D Rendering

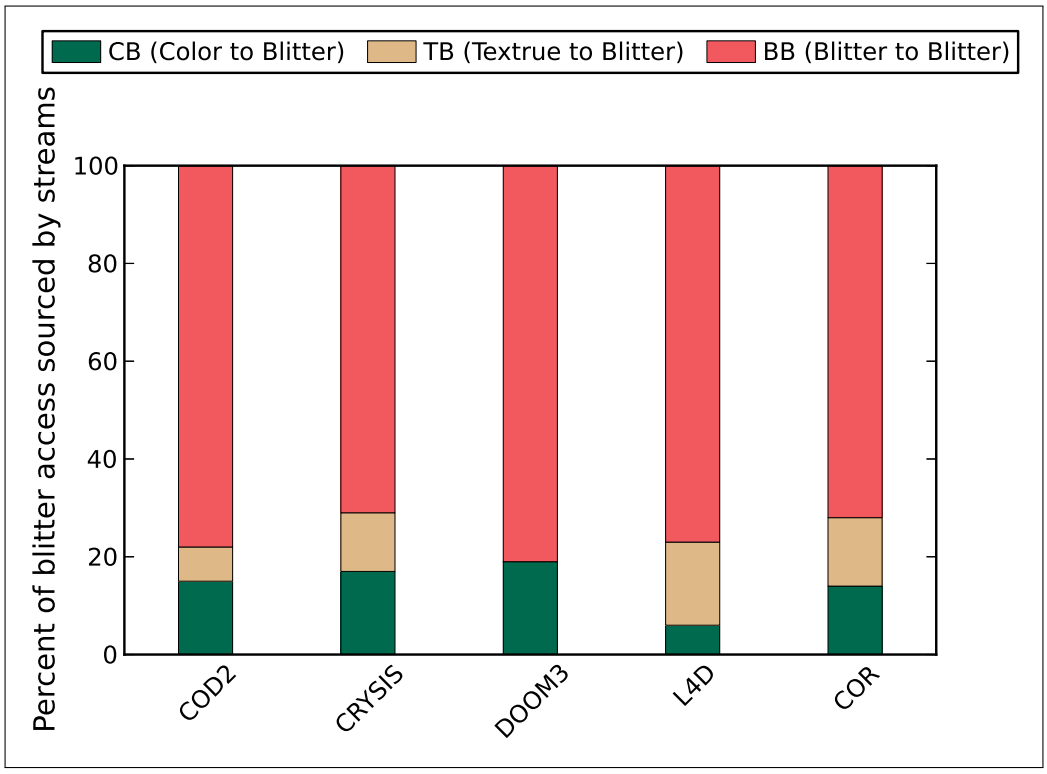


Figure 2.12: Color and texture to blitter inter-stream reuse

Figures 2.12 and 2.13 show the percentage of blitter and texture LLC accesses sourced by the C, Z, T, and B streams. These data are collected by extending each LLC block with a three bit stream-id. Whenever a block is touched by a stream other than its last accessing stream, the stream-id is updated to reflect the change. Every time such transition is detected, it is recorded in a reuse counter. Five applications (COD2, Crysis, DOOM3, L4D, and COR) exhibit C and T to B reuse. For COD2, Crysis, DOOM3, L4D, and COR, 22%, 29%, 19%, 23%, and 28% B stream accesses are sourced by C and T streams, respectively. C, Z, B to T reuses are shown by a much larger set of applications (all applications except DOOM3, Quake4, and UT4). For many applications, majority of the T stream reuses come

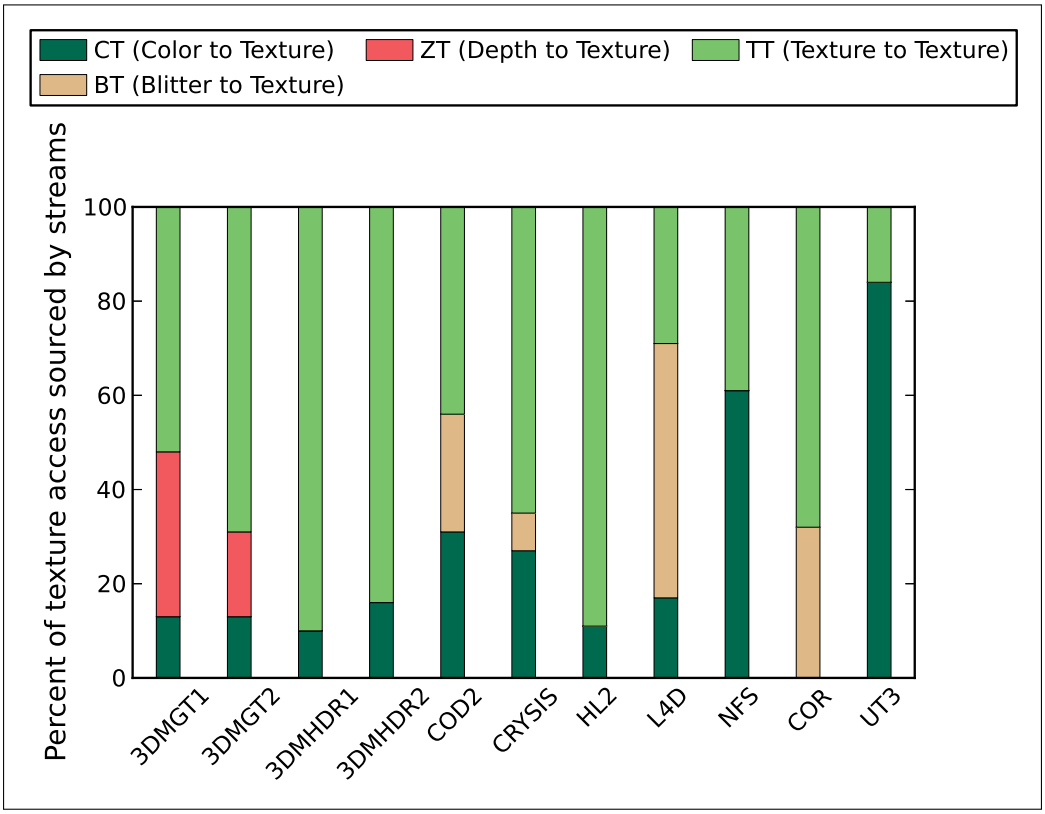


Figure 2.13: Color, blitter, and depth to texture inter-stream reuse

from the inter-stream sharing. For example, 61% of T stream accesses for NFS are sourced by the C stream alone. For L4D, the C and B streams source 17% and 54% T stream accesses, respectively. Similarly, for UT3, 84% T stream accesses are sourced by the C stream alone. For 3DMGT1 and 3DMGT2, the Z stream sources 35% and 18% T stream accesses, respectively. The color to texture reuses arise from the use of dynamically generated texture maps. The general technique for this is known as render-to-texture. The depth to texture reuses arise from the techniques used to generate dynamic shadow maps where the depth buffer contents are accessed by the texture samplers.

In summary, the results of this section indicate that in a heterogeneous environment, the performance of both CPU and GPU applications suffers from significant

degradation. Since the memory access streams sourced by the different rendering pipeline units have large working sets and varying reuse behavior, their efficient management is critical to both CPU and GPU applications. We also observe a significant amount of cross-stream reuses and variable sensitivity of GPU streams toward rendering throughput. These observations form the foundation of our contributions detailed in the subsequent chapters.

Chapter 3

Dynamic Reuse Probability-based Last-level Cache Management

This chapter presents our shared LLC management policy for the CPU-GPU heterogeneous processor. The central contribution of our proposal is a novel working set sampling technique that we employ to dynamically estimate the reuse probabilities of individual data streams coming from CPU and GPU. The estimated dynamic reuse probabilities are used to drive algorithms to manage the shared LLC. Compared to a state-of-the-art baseline with a 16 MB shared last-level cache, our proposal is able to improve the performance (frame rate or execution cycles, as applicable) of eighteen GPU workloads spanning DirectX and OpenGL game titles as well as CUDA applications by 12% on average and up to 51% while improving the performance of the co-running quad-core CPU workload mixes by 7% on average and up to 19%.

The rest of the chapter is organized as follows. Sections 3.1, 3.2, and 3.3 present motivational studies demonstrating that the gap left between the state-of-the-art LLC management policies and the Belady's OPT is significant and if the LLC hit

rate is improved, the GPU can gain significant performance. The dynamic reuse probability-based LLC management proposal is presented in Section 3.4. We discuss the existing related proposals for managing the LLC in Section 3.5. Section 3.6 presents experimental evaluation of our proposal. We conclude the chapter in Section 3.7.

3.1 Study on LLC Miss Savings

In this section, we evaluate a number of existing proposals in terms of the LLC read miss count and establish that there is a significant gap left between these proposals and the offline optimal policy due to Belady [4, 70]. We first briefly discuss the evaluated proposals in the following.

Baseline: The concepts of re-reference prediction value (RRPV) and re-reference interval prediction (RRIP) are introduced in [35]. The RRPV of a block maintains an inverse relation with the block's victimization priority. With an n-bit RRPV, the static re-reference interval prediction (SRRIP) algorithm statically assigns an RRPV of $2^n - 2$ to a block on insertion into the LLC. On a hit, the RRPV of the block is updated to zero. A block with RRPV $2^n - 1$ is selected as the victim. The baseline LLC follows the SRRIP algorithm when serving read misses, read hits, and LLC replacements. Additionally, the GPU can generate writes to blocks that are not resident in the LLC. Such a situation can arise for three reasons. First, the GPU can allocate and write to data in its internal color and depth caches without notifying the LLC and later it evicts such data from the internal caches to the LLC. Second, since the LLC does not maintain inclusion for GPU data, writebacks from GPU's internal caches may miss the LLC. Third, the shader cores bypass the private data

caches (in addition to evicting the target data cache block) when storing to global data to maintain coherence. As a result, all the evaluated policies must handle write misses and hits. Among the GPU data streams that are written to, color, depth, and shader data are the most important ones because these data are often reused by future reads. In both DirectX and OpenGL applications, dynamically generated color data can be reused as a texture for sampling [66]. Such texture data are usually referred to as dynamic texture data [26]. There are two ways to use color data as a texture map. First, a render target (containing color data) can be directly bound as a sampler resource and used as a texture map in DirectX applications. Second, the color data can be copied from the renderbuffer (of OpenGL) or render target (of DirectX) and transformed into a separate memory region before these data can be sampled as texture. This operation is known as blitting and the writes coming from the blitter to the LLC are also important from the viewpoint of future read reuses. Additionally, depth buffer contents can also be reused by the texture sampler for rendering shadow [26].

The color, depth, blitter, and shader write misses are inserted into the LLC at RRPV two (similar handling as the read misses). All other write misses bypass the LLC and go directly to the memory controllers. We found that for some heterogeneous mixes, this particular write miss policy degrades performance because the depth writes are not useful for all GPU applications. So, we further extend the write miss policy with a selective depth write bypass policy. The depth write bypass policy uses set dueling to decide if allocating depth write misses in the LLC is beneficial. It dedicates a group of LLC sets to always bypass depth write misses and another group of LLC sets to always allocate the depth write misses in the LLC. By comparing the relative number of read misses in these two groups, the depth write bypass

decision is made for every depth write miss to all LLC sets except these two groups. In our implementation, each of the groups has eight sampled sets per 1K LLC sets. More details on sampling-based set dueling can be found elsewhere [35, 36, 83]. The write hits do not change the RRPV of the blocks.

NRU: In the single-bit not-recently-used (NRU) replacement policy, each LLC block is provisioned with one replacement state bit. A read access to a block sets the bit. As a result of an access, if all bits in a set become one, the bits in all the ways except the currently accessed way are reset. The way with the lowest id and replacement state bit reset is the replacement candidate within a set. The write misses implement the same bypass policy as the baseline. The write misses that are allocated in the LLC are treated similarly as read misses. The write hits do not update replacement state.

DRRIP, TADRRIP: DRRIP [35] and TADRRIP [35] policies use the concept of re-reference prediction value (RRPV) to decide the insertion and victimization ages of a cache block. For an n bit RRPV, the DRRIP policy dynamically chooses between two insertion RRPVs, namely, $2^n - 1$ and $2^n - 2$ using a set-sample-based duel. On a read hit, the accessed block is always promoted to RRPV 0. A block with minimum way-id at RRPV $2^n - 1$ is chosen as the victim candidate. The TADRRIP policy treats the CPU cores and the GPU as different independent threads and lets each thread choose the best insertion RRPV (among $2^n - 1$ and $2^n - 2$) for read misses. The write miss and write hit policies are same as the baseline.

SHiP-mem: The SHiP [106] policy, instead of using set-sample-based duel, uses program counter (SHiP-PC), memory address (SHiP-Mem), or code path (SHiP-Iseq) signatures of load/store instructions to decide the insertion RRPV of the cache blocks. Since for the GPU, it is not always possible to associate a PC with an ac-

cess (e.g., accesses from the fixed function units), we evaluate the SHiP-mem variant. As proposed originally, we divide the physical address space into contiguous 16 KB regions. For each region, we learn the count of reuses by hashing a fourteen-bit region identifier (address bits [27:14]) into a 16K-entry table T of three-bit saturating counters. On an LLC hit to a block belonging to a particular region, the corresponding region counter is incremented by one. If a block gets evicted from the LLC without experiencing any reuse, the corresponding region counter is decremented by one. A block suffering a read miss is filled with an RRPV of three, if the corresponding region counter is zero; otherwise the block is inserted with an RRPV of two. The RRPV has an inverse relationship with the chance of future reuse and victimization priority. The write miss and write hit policies are same as the baseline.

SHiP-hybrid: We design a new variant of SHiP named SHiP-hybrid suitable for heterogeneous CMPs. This policy makes a small change in the SHiP-mem policy. For all CPU read misses, it executes the SHiP-PC policy [106] for deciding the insertion RRPV of a block. In other words, instead of using the fourteen-bit memory region identifier to index the 16K-entry saturating counter table T, it uses the lower fourteen bits of the program counter (hashed with the CPU core id) of the CPU load/store instructions. The GPU reads that miss in the LLC continue to use the fourteen-bit memory region identifier to index into the same saturating counter table T because it is not possible to associate program counters with a large number of GPU accesses that come from the fixed-function hardware (texture sampler, color blender, depth test unit, etc.).[1] In summary, SHiP-hybrid uses SHiP-PC for CPU reads and SHiP-mem for GPU reads. The write miss and write hit policies are same

[1] This is also the reason why it is not possible to have an effective GPU implementation of the other existing proposals (such as SDBP [50]) that rely on the program counters associated with the LLC accesses.

as the baseline.

OPT, OPT+Bypass, Baseline+Bypass: The OPT policy implements Belady's MIN replacement algorithm [4, 70] extended to handle both read and write misses. We also evaluate an optimal bypass policy running in conjunction with OPT and Baseline. In these policies, if the next-use distance of an incoming new block belonging to the GPU is larger than the next-use distances of all the blocks in the target LLC set, the block is not allocated in the LLC. Note that CPU blocks cannot bypass the LLC because that would violate inclusion of CPU data. The OPT, OPT+Bypass, and Baseline+Bypass results cannot be generated online because they need future information. These results are generated by collecting an LLC access trace for each heterogeneous workload mix and functionally simulating the policies offline on the collected traces. As a result, the outcomes of these policies are bound to the specific ordering of the LLC accesses recorded in the traces.

Figure 3.1 shows the normalized number of LLC read misses for the LLC policies averaged over the 1C1G, 2C1G, and 4C1G heterogeneous workloads. The simulation environment and the workloads were discussed in the previous chapter. The shared LLC capacity is 16 MB. Each bar within a heterogeneous configuration shows the normalized LLC read miss counts for a policy. Each bar further shows the contributions coming from the LLC read misses suffered by the CPU cores and the GPU. All bars in each group are normalized to the baseline policy (the leftmost bar in each group). In general, the fraction of LLC misses coming from the GPU decreases with increasing CPU core count because the proportion of CPU misses increases. We make three important observations from these results. First, SHiP-hybrid is the best among the online policies we consider. On average, it saves 7%, 8%, and 11% LLC read misses compared to the baseline for the 1C1G, 2C1G, and 4C1G config-

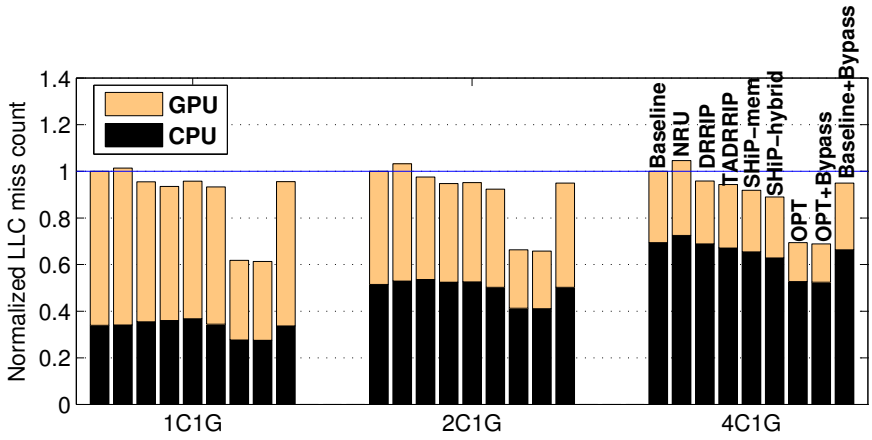


Figure 3.1: Normalized average read miss count.

urations, respectively. Second, there is a large gap between OPT and SHiP-hybrid indicating that there are significant opportunities for improvement. On average, the OPT policy saves 38%, 34%, and 30% LLC read misses compared to the baseline for the 1C1G, 2C1G, and 4C1G configurations, respectively. Among the CPU and the GPU read misses, the latter offers more opportunity for saving LLC read misses. Third, the bypass policies fail to reduce the LLC read miss count much indicating that for the heterogeneous workload mixes we consider in this study, an optimal bypass policy for GPU data from the viewpoint of minimizing the LLC read miss count is not particularly helpful. OPT+Bypass does not offer any additional benefit over OPT. The Baseline+Bypass policy fails to beat the SHiP-hybrid policy. We explore more aggressive GPU read miss bypassing in the later part of this chapter.

3.2 GPU Performance with Ideal LLC

The GPU architecture is known to have an inherent capability of hiding memory access latency due to the presence of a large number of ready thread contexts. Therefore, it is important to understand how important it is to save GPU LLC

misses. To quantify how sensitive the GPU performance is to the LLC miss count, we gradually make the LLC ideal for the GPU. We simulate the 1C1G configuration and gradually convert the GPU LLC misses to hits (except the compulsory misses). We conduct the following five sets of experiments for the heterogeneous mixes involving the 3D scene rendering workloads. First, we convert all non-compulsory color misses to LLC hits. Second, we treat all non-compulsory color and texture misses as LLC hits. Third, we treat all color, texture, and depth accesses to the LLC as hits (except the compulsory misses). Fourth, all color, texture, depth, and blitter accesses to the LLC are treated as hits provided they wouldn't lead to compulsory misses. Finally, all non-compulsory LLC misses from the GPU are converted to LLC hits. In all cases, all other accesses, including the accesses from the CPU core, are treated according to the baseline policy.

Figure 3.2 shows the progressive speedup (in terms of frame rate) observed by the 3D rendering applications as color (C), texture (C+T), depth (C+T+Z), blitter (C+T+Z+B), and all GPU non-compulsory misses in the LLC are converted to hits (stacked improvement from bottom to top in each bar). The overall speedup ranges from 15% (S4) to 145% (S10), averaging at 63%. Most of the benefits come

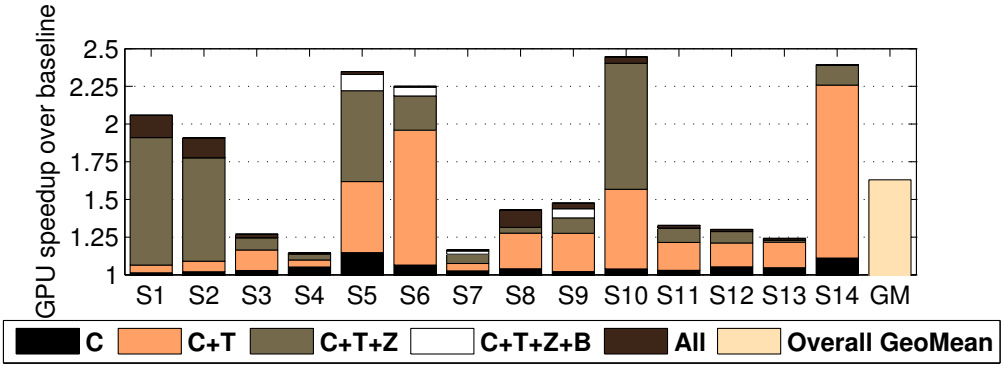


Figure 3.2: Potential improvement in frame rate. The baseline policy was introduced in Section 3.1.

from making texture and depth accesses ideal. Making color accesses ideal improves performance by more than 5% in S4, S5, S6, S12, S13, and S14, while only S5, S6, and S9 show more than 5% performance-sensitivity to the blitter access latency. Only S1, S2, and S8 enjoy more than 5% performance improvement when the LLC is made to behave ideally for the GPU streams other than C, T, Z, B. This additional improvement results primarily from elimination of the vertex misses. Overall, these results indicate that the GPU performance has widely varying sensitivity to the access latency of different data streams, particularly color, texture, depth, and blitter. Saving the LLC misses to these data streams can significantly improve the GPU performance for several workloads.

For the 1C1G heterogeneous mixes involving the CUDA applications, we study the impact on the performance of these applications when all non-compulsory LLC misses from the GPU are treated as LLC hits. Table 3.1 lists the observed speedup (over the baseline) in these applications. These results confirm that saving LLC misses can significantly improve the performance of these applications.

Table 3.1: Speedup of CUDA applications with ideal LLC

S15	S16	S17	S18
1.22	1.12	1.26	2.82

3.3 Selective LLC Bypass of GPU Read Misses

We have already shown that an optimal GPU read miss bypass policy with the goal of minimizing the overall LLC read miss counts is not particularly helpful (see Figure 3.1). In the following, instead of minimizing the overall LLC read miss count,

we study the performance impact of very aggressive GPU read miss bypass. This study is motivated by the fact that the GPU architecture can effectively hide the impact of a large volume of LLC misses resulting from aggressive read miss bypass. We conduct four experiments where we progressively increase the GPU read miss bypass percentage from 25% to 100%. Figure 3.3 shows the average speedup relative to the baseline for the CPU and the GPU workloads observed in these experiments. The CPU performance does not show any improvement until the GPU read miss bypass rate reaches 100%. At this point, the CPU performance improves, on average, by 5% in the 1C1G configuration and by only 1% in the 2C1G and 4C1G configurations. The CPU performance improvement drops drastically in the 2C1G and 4C1G configurations due to heavy congestion in the memory controllers caused by the aggressive GPU read miss bypass. The GPU performance, as expected, progressively suffers as the bypass rate increases. At 100% bypass rate (which is the only bypass rate useful for improving the CPU performance), the average loss in the GPU performance is 7%, 7%, and 9% in the 1C1G, 2C1G, and 4C1G configurations, respectively. In the 4C1G configuration, due to severe shortage of memory bandwidth resulting from aggressive bypass, the performance loss is more compared to the other two configurations. Overall, these results do not show much promise for an LLC management policy that relies on aggressive GPU read miss bypass. We note that this inference is different from what has been shown in a prior study involving GPGPU applications only [71]. We attribute this difference to a wider variety of GPU applications considered in our study.

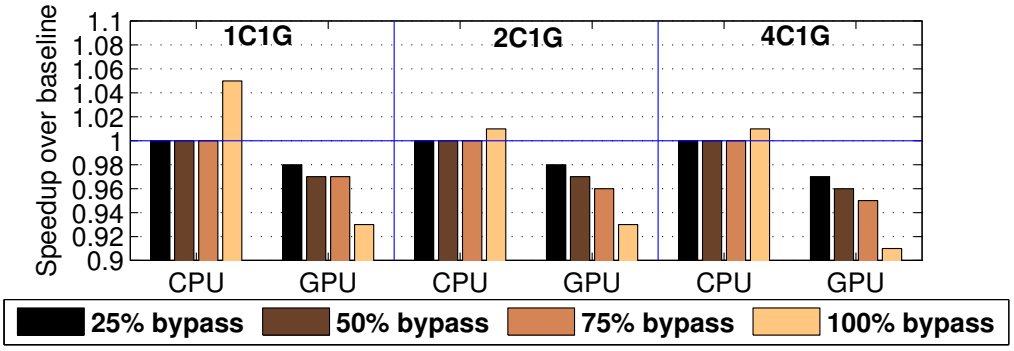


Figure 3.3: Speedup with random GPU read miss bypass for a 16 MB LLC. The baseline policy was introduced in Section 3.1.

3.4 Dynamic Reuse Probability for LLC

The design of an LLC management policy can be decomposed into four distinct sub-policies, namely, read miss policy, write miss policy, write hit policy, and read hit policy. We discuss the design of each of these sub-policies in the following. The write miss, write hit, and read hit policies form the crux of our dynamic reuse probability-based LLC policy proposal. These sub-policies make use of a working set sample (WSS) cache, the central contribution of our proposal. This WSS cache plays a key role in estimating the dynamic reuse probability of different data streams. We begin our discussion by introducing the architecture of the WSS cache. We note that the estimated dynamic reuse probabilities can be used in many different ways to implement an effective LLC management proposal. Our design assumes the existence of two replacement state bits per LLC block. These two bits can be thought of as age bits and we will refer to them as the RRPV bits, as in the baseline policy. The sub-policies modulate the RRPV bits. The victim selection algorithm is same as SRRIP. Although the discussion of our proposal revolves around the most general commercially available CMP architectures involving both CPU and GPU

cores, the general idea of the WSS cache can be employed to implement various types of optimizations in the LLC of discrete GPU parts as well as multi-core parts involving CPUs only.

3.4.1 Working Set Sample Cache

Our proposal employs a working set sampling technique to estimate the read reuse probability of all accesses that come to the LLC from CPU as well as GPU. For this purpose, we architect a small working set sample (WSS) cache shown in Figure 3.4. The WSS cache is a traditional set-associative cache.

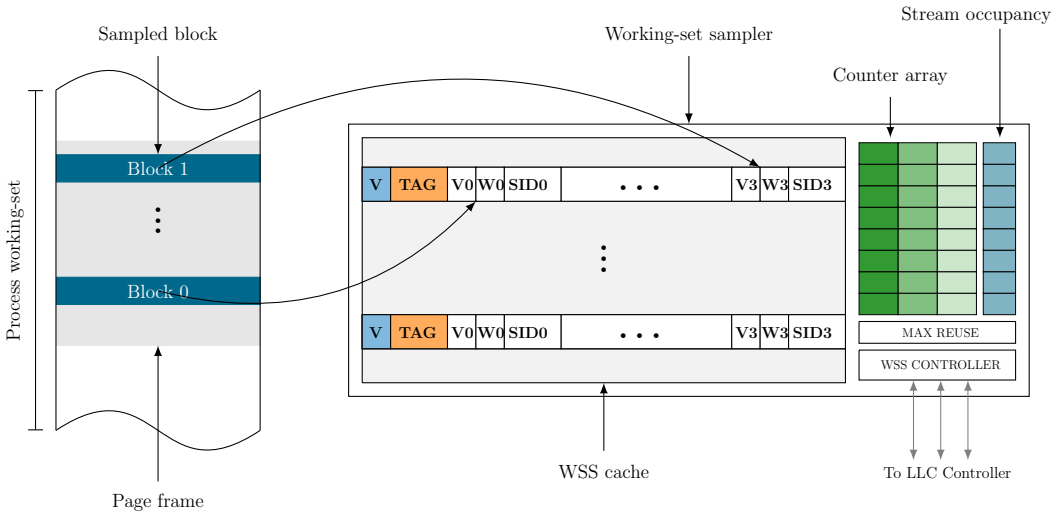


Figure 3.4: Organization of the WSS cache and the associated controller and external states such as the counter and occupancy arrays

Each entry of the cache tracks a few selected blocks in a sampled page. As a result, each WSS cache entry contains a page tag. To simplify the tracking of the sampled blocks in the sampled page, our design tracks every k^{th} block of the page where k is a design parameter. For each tracked block, we maintain the stream it belongs to and we consider the following stream categories: CPU, color, depth,

static texture (or simply texture), dynamic texture, blitter, shader, and the rest clubbed into one category. The CPU stream is further partitioned based on the originating CPU core. An LLC access is said to belong to a certain stream if the access originates due to a miss in an inner-level cache dedicated to that stream. For example, the shader stream arises from the misses in the shader cores' private caches. Additionally, for each tracked block, there is a valid bit (V) and a write bit (W). The W bit specifies if the block has been written to, but yet to be consumed by a subsequent read. A subsequent read reuse to such a block resets the W bit. Thus, each tracked block needs just six bits of state: four bits to encode the stream id (for a 4C1G CMP, we need to encode eleven different streams) and two bits for the V and W states. Therefore, if a physical page contains N blocks, each WSS cache entry needs a page tag, an entry valid bit, and $6(N/k)$ bits to track the sampled blocks. Figure 3.5 shows a typical WSS cache entry.

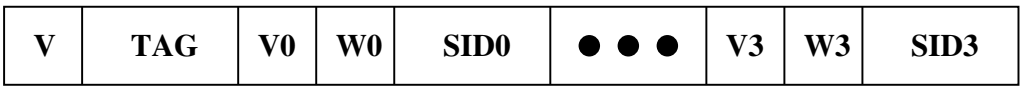


Figure 3.5: A WSS cache entry for $N = 64$ and $k = 16$. V is the entry valid bit, while V0-V3 are the valid bits for the four tracked blocks in the page. W0-W3 are the write bits of the tracked blocks and SID0-SID3 are the stream ids of the tracked blocks. TAG is the page tag of the entry.

On every LLC access, the WSS cache is looked up in parallel. On a WSS cache miss, an invalid WSS cache entry is allocated. If there is no invalid WSS cache entry in the target WSS cache set, the access bypasses the WSS cache. Since the purpose of the WSS cache is to estimate reuse probabilities, it is important to retain a sampled entry for a significantly large time-window so that the far-flung reuses can be captured. As a result, WSS cache replacements are usually disabled. Only if the accessing stream is found to have low representation in the WSS cache, a random

replacement policy is invoked. For this study, we allow WSS cache replacement if the accessing stream has less than 32 WSS cache entries. An entry is assumed to belong to the stream of the first valid tracked block in the entry. On a WSS cache hit, two situations may arise. If the accessing block turns out to be a tracked block (i.e., one of the k^{th} blocks in the page) and its entry is invalid, the entry is now marked valid and the appropriate state bits are updated. If the accessing block entry is valid and the access is a read, a reuse has been identified by the WSS cache. In this case, we increment a reuse counter to record this event.

Our design maintains two different arrays of reuse counters. The first array tracks, for each stream type, the count of write-to-read reuses captured by the WSS cache. We will refer to this array as the WR reuse counter array. The second array, referred to as the RR reuse counter array, is used to track read-to-read reuse counts for the streams. Our design also keeps track of the maximum among all reuse counters indicating the maximum reuse enjoyed by any stream during a phase of execution. We will refer to this as MAX_REUSE. In addition to the reuse counter arrays, our design also maintains a write access (WA) counter for each stream. This counter tracks, for each stream, the number of LLC writes captured by the WSS cache and is used to calculate the write-to-read reuse probability of a stream (which is the ratio of the WR counter of a stream to the WA counter of the stream). The static and dynamic texture streams do not need the WA counters because these are read-only streams (texture samplers only read texture data and never modify them). Figure 3.6 shows the flow for looking up the WSS cache on an access from stream S for a block which is the m^{th} tracked block in a page with tag P.

We define a WSS cache epoch to be a time-window over which the LLC receives 512K read accesses. At the end of each epoch, the WSS cache is invalidated and

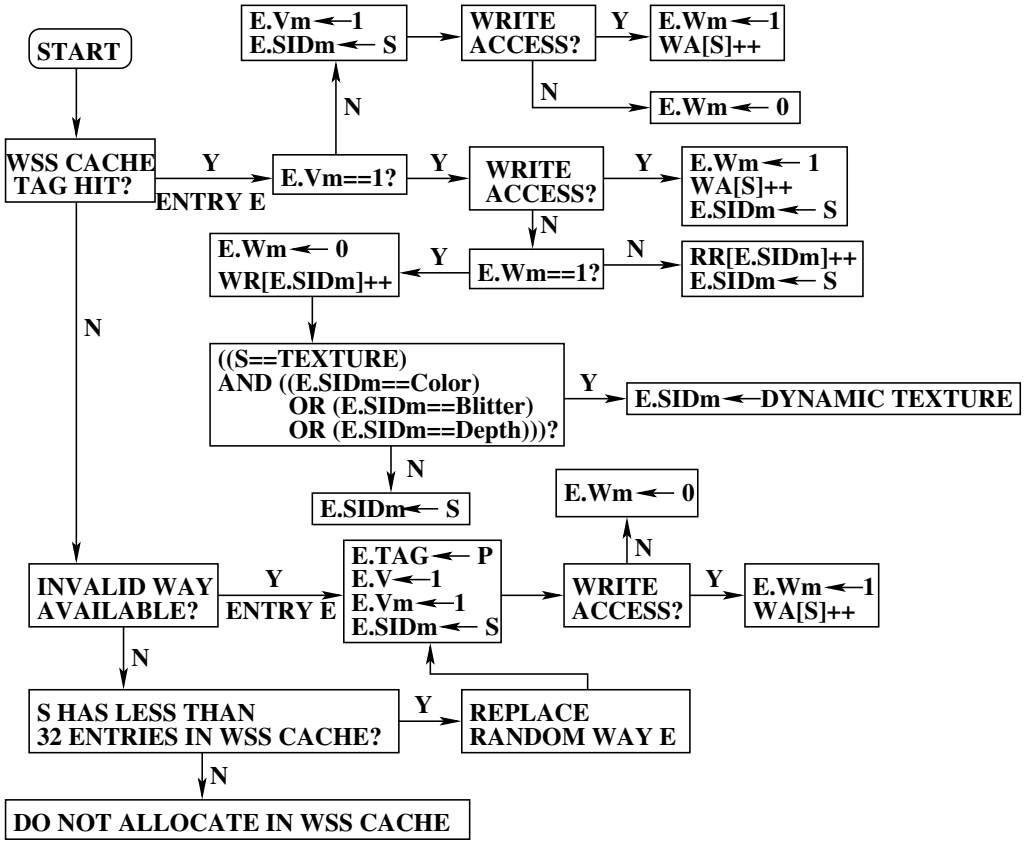


Figure 3.6: Access/Update protocol flow for the WSS cache and the accompanying reuse and access counters. Vm, Wm, and SIDm correspond respectively to the V, W, and stream id of the m^{th} tracked block. The MAX_REUSE register is not shown.

all the reuse and access counters are halved so that we can capture phase changes.

Next, we present the proposed sub-policies.

3.4.2 Read Miss Policy

The responsibility of the read miss policy is to decide the RRPV of the block being filled in the LLC. We synthesize our read miss policy by borrowing from the vast body of existing research that deals with LLC insertion policies on read misses discussed in Section 3.5 and evaluated in Section 3.1. While SHiP-hybrid is an

attractive design option, we observe that the large memory footprints of the GPU applications cause interference in the saturating counter table T while executing SHiP-mem for GPU read misses. As a result, our read miss policy does not use the SHiP-mem component of the SHiP-hybrid policy. For CPU read misses, we continue to use the SHiP-PC policy for deciding the insertion RRPV. For GPU read misses, we use the simpler DRRIP policy [35], which employs a set-sampling-based duel to decide among insertion RRPV of two and three. It is important to note that we adopt only the insertion policy component of DRRIP and invoke it on GPU read misses. Our read hit policy proposal is discussed in the later part of this section. Since the saturating counter table T is not required by the GPU read misses, it is exclusively used by the SHiP-PC policy exercised by the CPU read misses. This read miss policy is seen to outperform SHiP-hybrid for the workloads where the GPU application has large memory footprint. Figure 3.7 presents a high-level depiction of the proposed read miss policy.

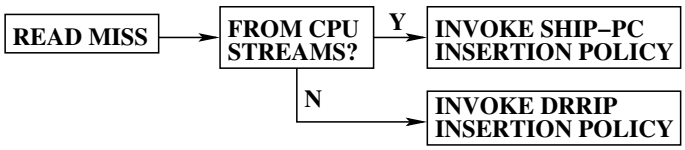


Figure 3.7: Read miss policy.

3.4.3 Write Miss Policy

The write miss policy is important only for the GPU because all CPU writes hit in the LLC due to inclusion of CPU data in the cache hierarchy. Recall that the baseline policy bypasses all GPU write misses except from color, depth, blitter, and shader streams and employs a selective bypass mechanism for depth write misses. The write miss policy must decide the insertion RRPV for the blocks that are allocated

in the LLC on write misses. We employ the write-to-read reuse probability of each stream for this purpose.

The write-to-read reuse probability of a stream can be calculated as the ratio between the corresponding WR reuse counter and the WA counter. On a write miss, if the decision is to allocate the block in the LLC, our design assigns an RRPV of zero if the block belongs to a high WR reuse stream. A high WR reuse stream is defined to be one that enjoys at least $MAX\_REUSE/3$ reuses or has a write-to-read reuse probability of at least 1/8. Additionally, if the WR reuse count enjoyed by the stream exceeds $MAX\_REUSE/2$ or has a write-to-read reuse probability of at least 1/8, the write miss policy identifies the block to be an important one and recommends that the block be pinned in the LLC. However, whether such a block will be finally pinned or not is decided by a per-stream set dueling because pinning write insertions does not always help and can hurt performance under heavy cache contention. The set dueling mechanism ear-marks two disjoint groups of sample sets for each stream (except static and dynamic texture because these streams are read-only). One group always follows the pinning recommendation, while the other group always ignores the pinning recommendation. Both the groups follow the remaining components of the policy unchanged. Based on the relative volume of read misses experienced by the two groups for a stream, the winning policy is decided for that stream and the rest of the sets follow the winning policy. In our implementation, each group for each stream has eight sampled sets per 1K LLC sets. We need four such disjoint group pairs representing the color, blitter, depth, and shader streams. A pinned block gets inserted into the LLC with RRPV zero. The RRPV of a pinned block is updated just like a normal block. When the RRPV of a pinned block reaches three, it is unpinned and its RRPV is reset to zero. Also, a pinned LLC block gets

unpinned when it receives a read reuse. In summary, pinned blocks get to spend more time in the LLC compared to a normal block.

Finally, if the stream that is having a write miss fails to qualify as a high WR reuse stream and has a write-to-read reuse count of zero with write access count of at least 128K, the block is inserted at RRPV three. All other write miss insertions happen at RRPV two. Figure 3.8 summarizes our write miss policy proposal.

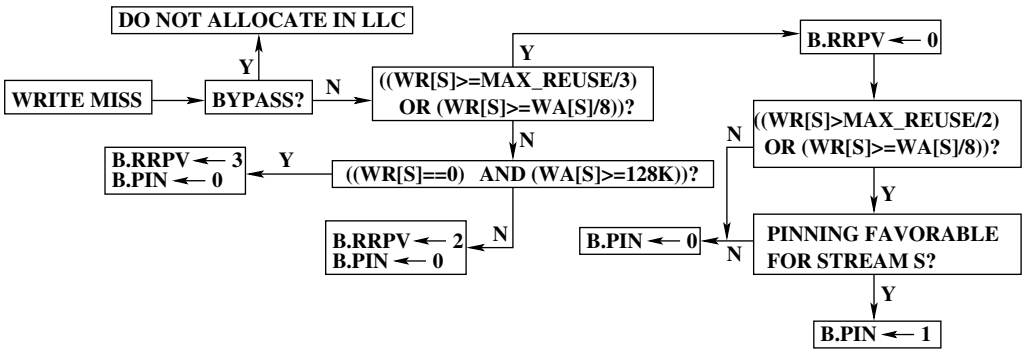


Figure 3.8: Write miss policy for a block B coming from stream S.

3.4.4 Write Hit Policy

The write hit policy is similar to the write miss policy in the sense that it attempts to give extra protection to the block receiving the hit if the block belongs to a high WR reuse stream. Also, all such high WR reuse stream blocks are recommended for pinning on a write hit. The goal of such a recommendation is to save write bandwidth at the memory controllers. For this purpose, we define a high WR reuse stream to be one which has received at least $MAX\_REUSE/2$ reuses or its write-to-read reuse probability is at least 1/16. A write hit to a block belonging to such a stream promotes the block to RRPV zero and pins the block; otherwise the block's RRPV is left unchanged and the pin state of the block is cleared. Again, we note

that pinning is only a recommendation from the write hit policy and the final pinning decision comes from a set duel already discussed. Since write hits can be experienced by the CPU streams as well, we need additional four pairs of set sample groups for deciding the favorability of pinning for the CPU cores in a 4C1G configuration. This write hit policy, which is congestion-oblivious, hurts performance if the set receiving the write hit is congested.

We find that for a congested set, a block belonging to a high WR reuse stream fails to enjoy most of the far-flung write-to-read reuses inferred by the WSS cache. As a result, to guarantee that such a block can enjoy at least the near-term write-to-read reuses without increasing the set congestion, it is enough to give the block extra protection only if its current RRPV is three (i.e., currently a candidate for victimization). Our congestion-aware write hit policy sets the RRPV of a block receiving a write hit to two if the block's current RRPV is three and it belongs to a high WR reuse stream. The RRPV of any other block is left unchanged.

On a write hit, the congestion-oblivious or the congestion-aware write hit policy is executed based on a set duel. This set duel employs two groups of sampled sets shared by all streams, each group having eight sets per 1K LLC sets. One group always executes the congestion-oblivious write hit policy, while the other group always executes the congestion-aware write hit policy. Based on the relative volume of the read misses experienced by the groups, the winning policy is decided and the rest of the sets follow the winning policy. While the write miss policy can improve the performance of the GPU applications only, the write hit policy can be beneficial to both CPU and GPU applications. Figure 3.9 shows the congestion-oblivious and congestion-aware write hit policies.

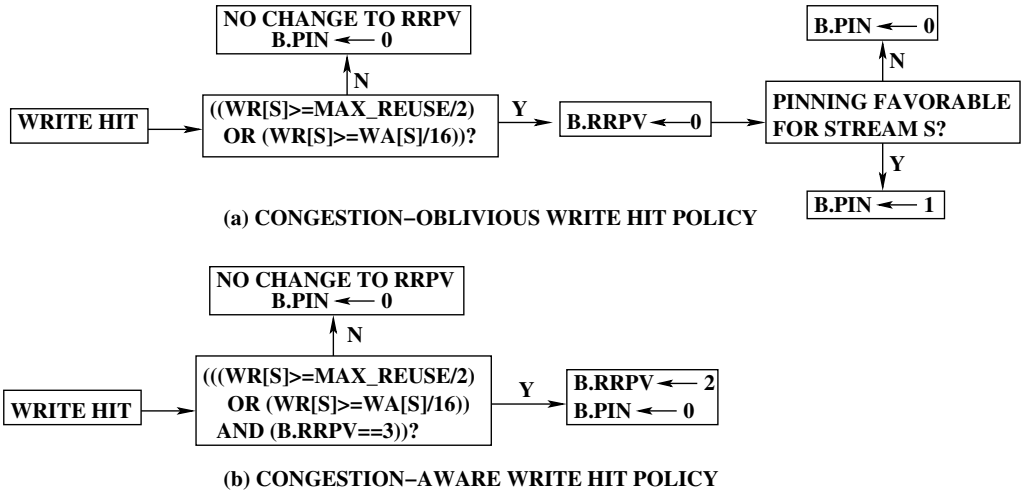


Figure 3.9: Write hit policies for a block B coming from stream S.

3.4.5 Read Hit Policy

Our read hit policy promotes the block receiving the hit to RRPV zero with one exception. We have observed that a large fraction of the dynamic texture blocks receive only one read access (the first texture sampler access to a block written to by color/blit/depth stream). Keeping such blocks longer in the LLC wastes space. We keep two counters to estimate the probability of the event that a dynamic texture block sampled by the WSS cache receives any reuse beyond the first access. If this probability is below 1/64, the dynamic texture block is demoted to RRPV three on its first read hit. If this probability is between 1/64 and half, the RRPV is set to two. In all other cases, the block is promoted to RRPV zero. To be able to implement this policy, the WSS cache entry needs to be extended by one bit for each sampled block to track the number of read reuses (only need to distinguish between zero or more reuses). Therefore, we need seven state bits per tracked block in a WSS cache entry. Figure 3.10 summarizes our read hit policy proposal.

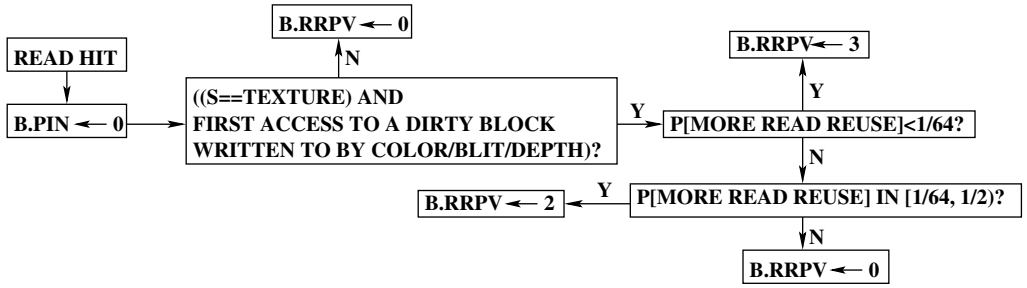


Figure 3.10: Read hit policy for a block B coming from stream S. P[E] denotes the probability of event E.

3.4.6 Storage Overhead

The primary storage overheads in our policy arise from the WSS cache, the extra state bits needed with each LLC block, and the saturating counter table T used in SHiP-PC. Our simulated system uses a page size of 4 KB and 48-bit physical addresses. Our design uses a 2K-entry WSS cache organized to have 128 sets and 16 ways. Therefore, the page tag of each WSS cache entry is 29 bits wide. We sample every eighth block in a sampled page. Therefore, each WSS cache entry tracks eight blocks leading to a total entry size of 86 bits (one valid bit, 29-bit tag, eight blocks×seven state bits/block). Thus, the WSS cache overhead is about 22 KB. Each LLC block, in addition to the RRPV bits, needs a pin bit. Two more bits per LLC block track the following four states: color/blit/depth write state (necessary to identify the dynamic texture blocks which consume data written to by the color, blitter, or depth stream), dynamic texture blocks with zero reuse count, dynamic texture blocks with at least one reuse count, and none of these. Thus, three extra bits are needed per LLC block leading to a total overhead of 96 KB (on top of the existing RRPV bits) for a 16 MB LLC. The saturating counter table T has 16K entries with each entry being a three-bit counter. Thus, T is of size 6 KB. In addition to these, the reuse and access counters and the fill PC signature (needed

by SHiP-PC) stored with the LLC blocks in a few sampled sets (32 per 1K LLC sets) add negligible overhead. Overall, our policy requires 124 KB of extra storage which is less than a percentage of the bits in the data array of the 16 MB LLC.

3.4.7 Latency Considerations

The hit/miss policies decide the RRPV and the PIN bit of the block being accessed. The general decision procedure involves a set of short comparisons, all of which can be done in parallel. For example, the write miss policy involves the following independent comparisons: (WR[S] $\geq$ MAX_REUSE/3), (WR[S] $\geq$ WA[S]/8), (WR[S] == 0), (WA[S] $\geq$ 128K), and (WR[S] $>$ MAX_REUSE/2). Note that whether pinning is favorable or not can be evaluated by another independent comparison between the read miss counters of the two set dueling groups. The comparison outcomes can then be combined using a couple of independent short multiplexers to assign the final values to RRPV and PIN. These comparators and multiplexers add negligible area overhead to the LLC controller logic. In the case of a hit (read or write), the critical path through this computation can be comfortably overlapped with the latency of data read-out from the LLC data array, which has a latency of three cycles (out of the total ten-cycle LLC lookup latency). In the case of a miss (read or write), the critical path is overlapped with the data write (i.e., fill) latency into the LLC data array. We assume the data write latency to be same as data read latency. Although the RRPV and PIN updates are overlapped with data array access, this implementation holds up the LLC tag array for the entire duration of ten cycles. As a result, our simulations pessimistically model a ten-cycle delay for all LLC tag lookups, part of which is overlapped with data access making the

overall critical path through the LLC ten cycles. The WSS array is looked up in parallel with the LLC tag array. Due to the small size of the WSS array, its lookup finishes much earlier than the LLC tag lookup.

3.5 Related Work

In this section, we discuss the contributions related to the management of LLCs in general-purpose CPUs, heterogeneous CMPs, and discrete GPUs.

3.5.1 LLC Management in CPUs

Dynamic insertion policy (DIP) adaptively inserts a block into the LLC at the LRU or the MRU position depending on the outcome of a set-sampling-based duel between LRU insertion and MRU insertion policies [83]. On a cache hit, a block is always upgraded to the MRU position. The replacement policy always victimizes the block at the LRU position. This algorithm tries to eliminate the single-use blocks from the LLC as early as possible without disturbing the rest of the contents of the LLC. A subsequent proposal has shown how to employ this policy in a shared LLC of a multi-core processor so that each thread can choose the best insertion policy [36]. A decision-tree based insertion age inference algorithm has also been proposed [49].

The concepts of re-reference prediction value (RRPV) and re-reference interval prediction (RRIP) are introduced in [35]. The RRPV of a block maintains an inverse relation with the block's victimization priority. With an n-bit RRPV, the static re-reference interval prediction (SRRIP) algorithm statically assigns an RRPV of $2^n - 2$ to a block on insertion into the LLC. On a hit, the RRPV of the block is updated to zero. A block with RRPV $2^n - 1$ is selected as the victim. The dynamic re-reference

interval prediction (DRRIP) algorithm dynamically chooses between two insertion RRPVs, namely, 2^n-2 and 2^n-1 based on the outcome of a set-sampling-based duel. Thread-aware DRRIP (TADRRIP) applies the technique proposed in [36] to allow multiple independent threads to execute DRRIP in a multi-core shared LLC. Recent proposals exploit signature-based hit prediction (SHiP) to improve the RRIP policies by using the program counters (SHiP-PC), memory addresses (SHiP-mem), or code path signatures (SHiP-Iseq) of the load/store instructions [106]. These variants of RRIP differ only in the way they assign a victimization priority to a block at the time of insertion into the LLC, but they handle hits and replacements in the same way. Explicit prediction of reuse distance [47], estimation of approximate next-use distance [69], and estimation of protection distance [17] have also been used to improve LLC performance.

Algorithms to partition the LLC among the referenced and non-referenced blocks and grow/shrink these partitions based on the dynamic demand have been explored [51]. Also, there have been LLC management proposals designed based on the observation that the LLC read misses originating from loads are more critical than those originating from stores [48]. Use of a small Bloom filter to capture a subset of the recently evicted blocks and algorithms to offer higher protection to a subset of such blocks which are accessed soon have been explored [91].

Another class of LLC management policies attempt to predict the dead blocks in the cache and victimize them early. The dead block prediction algorithms correlate the program counters of the load/store instructions with the death of the cache blocks that these instructions touch [29, 50, 52, 53, 59, 65]. Probabilistic escape LIFO is a light-weight dead block prediction technique that does not require the program counter signature and relies only on the fill order of the cache blocks within

a cache set [8]. Reuse pattern-based simple hints from the inner levels of the cache hierarchy in conjunction with a clever partitioning of the address space have also been used to effectively identify the dead and live LLC blocks [7, 22].

Algorithms have been proposed to explicitly partition the shared LLC among the competing threads of a multi-core processor. The utility-based cache partitioning (UCP) algorithm carries out a coarse-grain partitioning of the LLC by dynamically assigning a number of ways to each thread [84]. The promotion/insertion pseudo-partitioning (PIPP) policy improves UCP by designing smart insertion and promotion policies for cache blocks within each partition [107]. Subsequent proposals such as Vantage [90] and PriSM [68] eliminate the limitations of way-grain partitioning and allow each thread to have an arbitrary fine-grained partition. A recent proposal designs dynamic partitioning policies for the LLC using a model that can predict the application slowdown caused by the destructive interference in the LLC shared by multiple CPU cores [100]. Since the existing cache partitioning techniques treat the streams or threads as independent, these techniques cannot be applied directly to the 3D graphics streams, which have significant inter-stream data sharing (e.g., between render target and texture sampler access streams [21]). Our policy, instead of carrying out an explicit partitioning, induces implicit fine-grain partitions among the streams by estimating per-stream dynamic reuse probability and allocating more space to the streams that are likely to enjoy more reuses.

3.5.2 Managing LLC in Heterogeneous CMPs

The TLP-aware policies (TAP) for managing the LLC in a heterogeneous CMP extend RRIP and UCP policies to take into account GPU accesses to the shared

LLC [62]. Since these policies only target GPGPU-style scientific computing workloads running on the GPU, it is enough to understand how the shader cores react to changing LLC allocations ignoring the performance of the rest of the graphics pipeline. As a result, these policies (TAP-RRIP and TAP-UCP) sample two shader cores and allow the accesses coming from these two cores to follow LRU and MRU insertion policies in the LLC. Based on the performance difference of these two sampled cores, the proposal decides if the executing GPU application is LLC-sensitive. Accordingly, the proposal makes modifications to the RRIP and UCP policies. To apply this proposal to 3D scene rendering applications, it is necessary to sample two rendering pipelines consisting of two distinct slices of several fixed function units as well as two shader cores. To observe any difference in performance between the two sampled rendering pipelines, enough work must be done by the pipelines; the difference in performance impact due to LRU and MRU insertions takes time to manifest, particularly in the presence of large reuse distances so that even the MRU-inserted blocks may get replaced before getting reused. We observe that this time window is typically equivalent to processing of a few batches of polygons. However, to satisfy ordering requirements between two consecutive batches, the processing of a fresh batch cannot begin until the processing of the last batch is completed. Due to this implicit global synchronization between the parallel rendering pipelines inside the GPU, the performance difference between the sampled pipelines cannot be accumulated across batches. As a result, sampling different pipelines and observing how they react to different LLC policies, as TAP does, is not helpful for 3D scene rendering workloads. In contrast, our proposal improves the performance of the entire graphics pipeline by basing the LLC policy decisions on estimated dynamic reuse probabilities.

Another proposal (HeLM) considers not allocating a fraction of the GPU data (coming from GPGPU-style scientific computing workloads running on the GPU) in the shared LLC if it is estimated that the CPU workload is LLC-sensitive and the GPU workload can tolerate LLC miss latency [71]. The degree of latency tolerance of a GPU workload is determined by taking into account the number of shader thread contexts ready to be scheduled at any point in time. A larger number of ready contexts usually offers bigger latency tolerance. The exact relationship between the degree of latency tolerance and the volume of ready thread contexts is estimated by sampling two shader cores and letting them bypass their misses at two different rates (one low and one high). Since the performance difference between two widely different bypass rates becomes visible much faster than LRU/MRU insertions (as is done in TAP), we find that the HeLM proposal can be adopted to the rendering pipelines more effectively even in the presence of the implicit synchronization between consecutive polygon batches. However, since HeLM relies on the number of ready shader thread contexts for determining the degree of latency tolerance, such a technique is expected to work only for those GPU workloads that exercise only the shader cores of the GPU and no other parts of the rendering pipeline.

To the best of our knowledge, ours is the first proposal that considers optimizations to the shared LLC of a heterogeneous CMP executing 3D graphics as well as GPGPU workloads in the presence of co-running CPU workloads.

3.5.3 LLC Management in Discrete GPUs

The graphics stream-aware probabilistic caching proposal discusses algorithms for improving the LLC performance in discrete GPUs [21]. These algorithms exploit

semantic information regarding 3D graphics streams and modulate the RRPV of a block based on the reuse behavior of the stream it belongs to. Since the reuse behavior is estimated by observing a few sampled LLC sets, the estimation is not accurate and changes depending on the accuracy of the replacement policies of the sampled LLC sets. In this paper, we estimate reuse probabilities by working set sampling, which is not affected by the implementation of the LLC.

Large number of proposals have explored policies to improve the efficiency of the internal caches in the discrete GPUs. These include shader cores' L1 cache bypass policies for GPGPU-style scientific computing workloads [11], shader cores' L1 cache allocation policies based on a certain priority assignment to the shader threads executing GPGPU-style scientific computing workloads [64], and various optimization on the texture cache architecture [12, 13, 25, 31, 32, 104]. Additionally, there have been proposals exploring shader thread scheduling mechanisms that are shader cores' L1 cache performance-aware [38, 40, 45, 61, 63] or memory divergence-aware [88]. DRAM scheduling techniques to minimize the main memory access latency variation across the shader threads within a scheduling group (called a warp in Nvidia GPUs) have also been proposed [6]. Some of these memory hierarchy-aware scheduling techniques may help address shader thread-induced contention in large LLCs in applications that make heavy use of the shader cores to process large amounts of global data with irregular access patterns. However, in the 3D scene rendering applications, large volumes of data originate from fixed-function hardware and the shader threads typically operate on contiguously allocated pixel fragments (during pixel shading) and vertex attributes (during vertex shading) ruling out the possibility of conflict-induced loss of locality in shader data.

3.6 Simulation Results

In this section, we evaluate our dynamic reuse probability (DRP)-aware policy proposal in terms of performance improvement and LLC miss savings for a 16 MB LLC. We begin the discussion by presenting the average (geometric mean) speedup achieved by our proposal and the SHiP-hybrid policy, which we have designed to represent a version of the SHiP proposal suitable for a CPU-GPU heterogeneous environment. This comparison is shown in Figure 3.11. The speedup averages are shown separately for the CPU and the GPU workloads, the average being computed over all eighteen heterogeneous mixes. For the 1C1G configuration, our DRP-aware policy improves average GPU performance by 8%, while SHiP-hybrid achieves an average improvement of only 3%. None of the policies, however, is able to improve the CPU performance much (less than 2% improvement). For the 2C1G configuration, the DRP-aware policy improves average GPU performance by 9%, while SHiP-hybrid exhibits an average speedup of 5%. The improvement in the average CPU performance is 3% and 4%, respectively for the SHiP-hybrid policy and the DRP-aware policy. For the 4C1G configuration, the DRP-aware policy improves average GPU performance by 12% and SHiP-hybrid is able to improve the GPU performance by 7%, on average. For this configuration, our proposal lags a couple of percentages behind the SHiP-hybrid policy for the CPU performance (7% improvement in our proposal compared to 9% in SHiP-hybrid); our proposal sacrifices some CPU hits to improve GPU performance significantly for the 4C1G configuration. In general, as the CPU core count increases, both the policies offer better improvements compared to the baseline with our DRP-aware proposal staying reasonably ahead of the SHiP-hybrid policy for GPU performance.

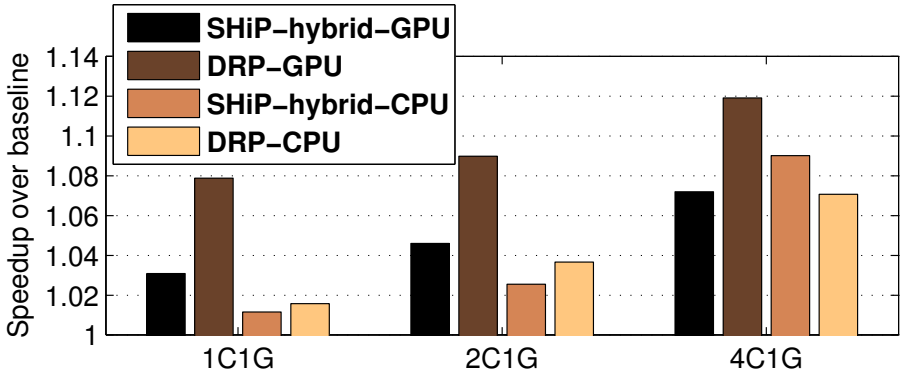


Figure 3.11: Average speedup comparison. The baseline policy was introduced in Section 3.1.

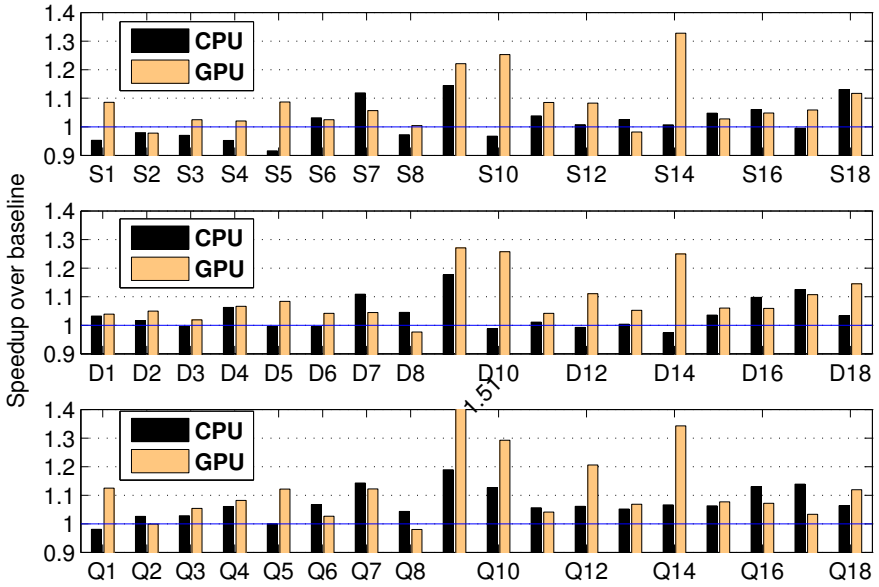


Figure 3.12: Speedup achieved by the DRP-aware proposal for the mixes. The baseline policy was introduced in Section 3.1.

Figure 3.12 presents the performance speedup achieved by our DRP-aware policy for each of the heterogeneous mixes. For each mix, we show the GPU and CPU speedup separately. The top, middle, and bottom panels show the results for the 1C1G, 2C1G, and 4C1G configurations, respectively. Across the board, the GPU performance improves significantly. Several workloads enjoy at least 10% improve-

ment in GPU performance, the maximum gain being 51% experienced by Q9. The improvement in CPU performance is much less, particularly for the 1C1G and 2C1G configurations. However, for the 4C1G configuration, several mixes enjoy more than 5% CPU performance improvement, the maximum gain being 19% experienced by Q9. In the 1C1G configuration, the CPU performance suffers a slowdown in some mixes because of back-invalidations induced by premature LLC replacement of CPU blocks.

To understand the source of the performance improvements, Figure 3.13 shows the normalized LLC read miss count for the baseline and our DRP-aware proposal. The results are normalized to the baseline policy. The top, middle, and bottom

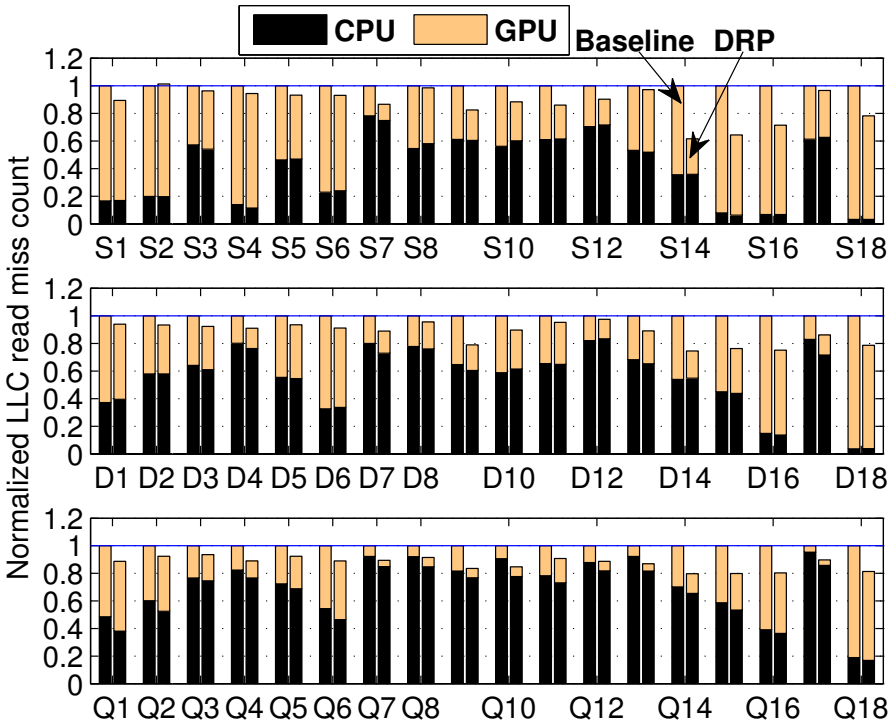


Figure 3.13: Normalized read miss count of the mixes. The baseline policy was introduced in Section 3.1.

panels show the results for the 1C1G, 2C1G, and 4C1G configurations, respectively.

Across the board, we see impressive LLC read miss savings achieved by the DRP policy. For the 1C1G and 2C1G configurations, the volume of CPU misses remains mostly unaffected, except for a few cases, most notably D17, which enjoys a significant reduction in the volume of CPU misses. Additionally, D4 and D9 show some reduction in the volume of CPU misses. Among the workloads that show more than 5% improvement in the CPU performance with DRP in the 1C1G and 2C1G configurations, S9, S15, S16, S18, and D16, do not show any noticeable reduction in the CPU read miss volume. The CPU workloads of these mixes benefit from an overall reduction in the LLC read miss count leading to lowered congestion and queuing delays in the memory controllers resulting in an improvement in the LLC miss latency. Each of these workloads enjoys at least 20% reduction in the total LLC read miss count. For the 4C1G configuration, the CPU read miss counts reduce significantly across the board (exceptions are Q3, Q5, Q16, and Q18). However, Q16 and Q18 experience significant improvement in the CPU workload performance due to reduced queuing delays in the memory controllers. Each of these two mixes enjoys at least 20% reduction in the total LLC read miss count. Turning to the GPU read misses, we observe that the DRP proposal is able to reduce the volume of these misses across the board. Overall, in all the configurations, a significant number of workloads enjoy at least 10% saving in the total LLC read miss count.

The DRP proposal exercises four sub-policies, namely, the read miss policy, the read hit policy, the write miss policy, and the write hit policy. In the following, we quantify the contribution of these sub-policies toward saving LLC read misses. Figure 3.14 summarizes the average savings in LLC read misses (averaged over eighteen mixes) for the 1C1G, 2C1G, and 4C1G configurations. In each configuration, the two leftmost bars correspond to the baseline and the SHiP-hybrid policies. The

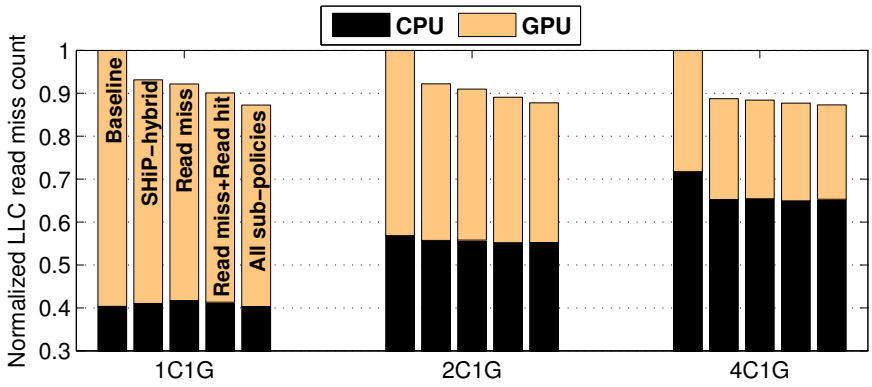


Figure 3.14: Normalized average read miss count.

next three bars quantify the gradual savings in the LLC read misses as we enable different sub-policies of our DRP proposal. The "Read miss" bar shows the effect of enabling the read miss sub-policy. The "Read miss+Read hit" bar shows the effect of enabling both read miss and read hit sub-policies. The last bar in each group shows the effect of enabling all the sub-policies i.e., this bar quantifies the average normalized LLC read miss count of our DRP proposal. Since the write miss and the write hit sub-policies are similar in nature, we do not show their benefits separately. The combined benefit offered by these two sub-policies can be seen in the difference between the rightmost bar and the "Read miss+Read hit" bar in each of the configurations. In the 1C1G and 2C1G configurations, on average, the volume of CPU misses remains almost unaffected. However, in the 4C1G configuration, there is a significant reduction in the volume of CPU misses compared to the baseline. Our DRP proposal, on average, saves 13%, 12%, and 13% LLC read misses in the 1C1G, 2C1G, and 4C1G configurations, respectively. It achieves significant savings in the GPU misses across the board. All the sub-policies exhibit important contributions to the overall LLC miss savings. We note that the Read miss sub-policy is slightly better than the SHiP-hybrid policy in the 1C1G and 2C1G configurations.

The SHiP-hybrid policy, on average, enjoys 7%, 8%, and 11% LLC miss savings in the 1C1G, 2C1G, and 4C1G configurations, respectively. For the 4C1G configuration, our DRP proposal saves 7% LLC read misses on average compared to the SHiP-hybrid policy if we consider only the GPU misses. These savings offer a significant advantage in GPU performance to the DRP-aware policy compared to the SHiP-hybrid policy for the 4C1G configuration, as already shown in Figure 3.11.

3.6.1 Comparison to Related Proposals

There have been two proposals, namely, TAP [62] and HeLM [71], for managing the shared LLC in heterogeneous CMPs. These proposals were briefly introduced in Section 3.5. We also pointed out that due to an implicit synchronization between the processing of consecutive batches of polygons in the 3D scene rendering applications, the TAP proposal loses its effectiveness in these applications. On the other hand, HeLM relies on aggressive bypassing of GPU read misses if the GPU application shows latency-tolerance through the presence of a good number of ready shader thread contexts.

Figure 3.15 shows the average speedup (over the baseline) of our DRP-aware proposal and HeLM. For each configuration, we show the average speedup achieved by the CPU and the GPU workloads separately. The left half of the results shows the average over all eighteen mixes, while the right half shows the average over only the mixes having CUDA applications. The left half shows that HeLM is not particularly effective for this set of applications (as already pointed out in Section 3.3). Only for the 4C1G configuration, it is able to improve the CPU performance by 6% while sacrificing slightly over 2% GPU performance. The average speedup for the CUDA

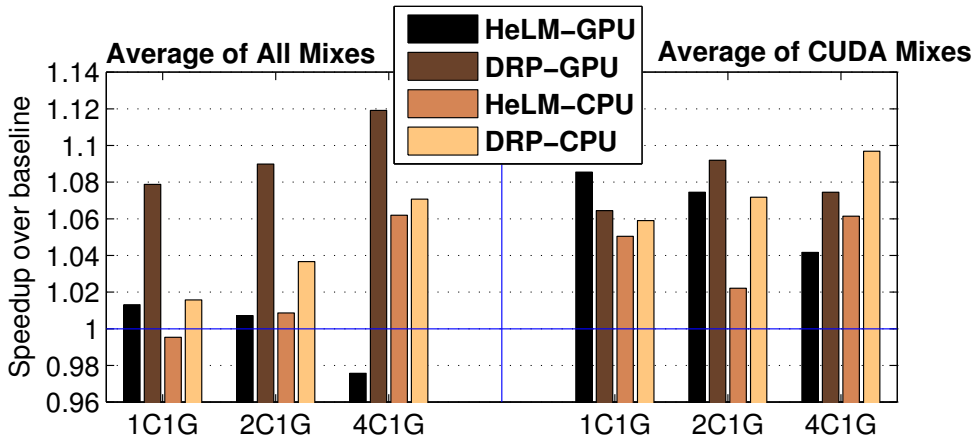


Figure 3.15: Speedup of HeLM and DRP. The baseline policy was introduced in Section 3.1.

mixes, however, confirms that HeLM can be effective for the GPU applications that make use of only the shader cores. This was the original scenario for which the HeLM policies were designed. HeLM identifies a GPU application as latency-tolerant by looking at the number of ready shader thread contexts. Such a mechanism is expected to work only for those GPU applications that exercise primarily the shader cores. On the other hand, the 3D scene rendering workloads exercise several fixed function units in addition to the shader cores. As a result, determining latency-tolerance of such applications requires more involved techniques. Nonetheless, our DRP-aware proposal still outperforms HeLM in all cases even for the CUDA mixes except for the GPU performance in the 1C1G configuration, where HeLM enjoys a nearly 9% speedup as compared to nearly 7% speedup achieved by DRP. With the increasing CPU core count, the bypass-induced congestion in the memory controllers begins to affect the benefits of HeLM for the CUDA mixes.

3.7 Conclusion

We have presented a novel LLC management policy for the emerging heterogeneous CMPs. Our proposal estimates the reuse probabilities of different access streams seen by the LLC and exploits these estimates to manage the blocks in the LLC. At the heart of our dynamic reuse probability estimation technique is a small working set sample cache, which retains a few blocks in a few sampled pages to learn the near-term and far-flung reuse probabilities. Our proposal saves 13% LLC read misses on average, improves the GPU workload performance by 12% on average, and improves the CPU workload performance by 7% on average in a CMP with four CPU cores and one GPU.

Chapter 4

QoS-guided Dynamic GPU Access Throttling

The focus of this chapter is the scenario where the GPU of a heterogeneous processor is used to execute 3D animation utilizing the entire rendering pipeline and at the same time the CPU cores are used to carry out general-purpose computing. Such a scenario arises in various real-world situations. For example, when the GPU is rendering a 3D animation frame, the CPU cores are typically engaged in preparing the geometry of the next frame requiring AI and physics computation. Also, in a high-performance computing facility, while the CPU cores do the heavy-lifting of scientific simulation of a certain time step, the GPU can be engaged in rendering the output of the last few time steps for visualization purpose [72, 96]. In this chapter, we present memory system management driven by the quality of service (QoS) requirement of the 3D scene rendering applications executing on the GPU along with applications on the CPU cores in such heterogeneous platforms. Our proposal dynamically estimates the level of QoS (e.g., frame rate in 3D scene rendering) of

the GPU application. Unlike the prior proposals, our algorithm does not require any profile information and does not assume tile-based deferred rendering. If the estimated quality of service meets the minimum acceptable QoS level, our proposal employs a light-weight mechanism to dynamically adjust the GPU memory access rate so that the GPU is able to just meet the required QoS level. The memory system resources, thus freed, is shifted to the co-running CPU applications. Detailed simulations done on a heterogeneous chip-multiprocessor with one GPU and four CPU cores running heterogeneous mixes of DirectX, OpenGL, and CPU applications show that our proposal is able to improve the CPU performance by 18% on average. This chapter's proposal does not impact the performance of the following two heterogeneous computing scenarios.

- The GPU executes 3D scene rendering applications that fail to meet the target QoS (frame rate) necessary for satisfactory for visual experience in certain phases. In these phases, out proposal does not find any opportunity of shifting memory system resources from the GPU to the CPU and leaves the performance of the applications unchanged.

- The GPGPU applications do not have any well-defined QoS target. The performance of the heterogeneous workload mixes containing GPGPU applications is not influenced by our proposal.

We address the performance issues of the heterogeneous workload mixes encapsulating these two scenarios in the next chapter. This chapter's focus is on the heterogeneous workload mixes containing 3D scene rendering applications that meet the target frame rate requirement.

The rest of the chapter is organized as follows. Section 4.1 presents a study demonstrating that despite degradation in the heterogeneous mode of execution, certain GPU applications perform above the required QoS limit for adequate visual experience and that the existing resource shifting algorithms are sub-optimal. Section 4.2 presents our QoS-guided memory management proposal. Sections 4.3, 4.4, and 4.5 present related work, simulation results, and conclusions, respectively.

4.1 Motivation

In this section, we first motivate the necessity to study memory system resource management techniques in CPU-GPU heterogeneous processors by showing that the co-running CPU and GPU applications in such systems suffer from significant performance degradation compared to when they are run standalone. These results were discussed in detail in Chapter 2 (see Figure 2.7). We reproduce a portion of these results here for reader's convenience. We observe that, even after suffering from a significant degradation in performance in the heterogeneous mode of execution, several GPU applications still continue to deliver a level of performance that is higher than necessary. When GPUs are used for rendering 3D scenes, it is sufficient for them to achieve a minimum acceptable frame rate. This relaxation is guided by the fact that due to the persistence of vision, human eyes cannot perceive a frame rate that is above a limit. This observation motivates us to explore mechanisms to dynamically shift memory system resources from the GPU to the CPU cores so that the GPU can just meet the minimum frame rate requirement. Prior studies have explored LLC bypassing as a technique to dynamically shift cache capacity from the GPU to the CPU cores in heterogeneous processors [71]. These studies rely

on the inherent latency hiding capability of the GPUs and assume that bypassing LLC won't degrade GPU performance. These studies, however, seem to overlook the impact of increased demand of the DRAM bandwidth due to a large number of GPU fills bypassing the LLC causing an increase in the volume GPU LLC misses. The last chapter has highlighted this fact demonstrating that aggressive LLC bypass by the GPU can degrade both CPU and GPU performance due to increased DRAM congestion. In this chapter, we further argue that compared to LLC bypassing, throttling the GPU access rate to the memory system is a more effective technique for shifting memory system resources to the co-running CPU applications.

To understand the implication of heterogeneous execution on the performance of the CPU and the GPU applications when they execute together and contend for the memory system resources, we conduct a set of experiments. In these experiments, the heterogeneous CMP has one CPU core and a GPU clocked at 4 GHz and 1 GHz, respectively (1C1G configuration). The shared LLC is of 16 MB capacity and there are two on-die single-channel DDR3-2133 memory controllers.[1] In the first of these experiments, we run a CPU job (SPEC CPU 2006 application) on the CPU core and keep the GPU free (standalone CPU workload execution). In the second experiment, we run a 3D animation job (drawn from DirectX and OpenGL games) on the GPU and keep the CPU free (standalone GPU workload execution). Finally, we run both jobs together to simulate a heterogeneous execution scenario.

Figure 4.1 shows the performance of the CPU job and the GPU job in the heterogeneous mode normalized to the standalone mode for fourteen such heterogeneous workload mixes (S1 to S14 from Table 2.4 in Chapter 2). Each workload mix contains one DirectX or OpenGL application and a SPEC CPU 2006 appli-

[1] Chapter 2 discusses our simulation environment in more detail.

cation. On average, both the CPU and the GPU lose 22% of performance when going from the standalone mode to the heterogeneous mode (see the GMEAN group of bars).[1] This loss in performance results from the contention for LLC capacity and DRAM bandwidth between the two types of applications running simultaneously. Prior studies have also observed large losses in performance due to memory system resource interference between the co-running CPU and GPU applications [3, 37, 46, 62, 71, 78, 85, 94, 95, 103].

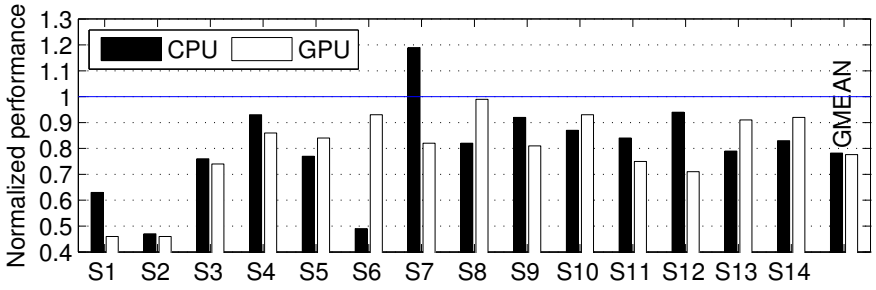


Figure 4.1: Performance of CPU and GPU in heterogeneous execution normalized to standalone execution. The y-axis shows the ratio of the standalone execution time to the heterogeneous execution time.

Even though a 3D scene rendering workload suffers from a large loss in performance when going from the standalone mode to the heterogeneous mode of execution, this loss may not be noticed by an end-user if the frame rate continues to be above the level required for visual satisfaction. Figure 4.2 shows the frame rates of the individual GPU applications belonging to the fourteen heterogeneous mixes for both standalone and heterogeneous modes of execution. We observe that even in the heterogeneous mode several GPU applications continue to deliver a frame rate that is comfortably above the 30 frames per second (FPS) mark, which is generally

[1] The CPU application in S7 enjoys a 19% improvement in performance in the heterogeneous mode compared to the standalone mode due to an unpredictable improvement in DRAM row-buffer locality.

considered to be the acceptable frame rate for visual satisfaction. Ideally, such applications should relinquish part of the memory system resources so that they can be utilized by the co-scheduled CPU applications. The challenge in designing such a dynamic memory system resource allocation scheme is two-fold. First, one needs to estimate and accurately project the frame rate of a GPU application. Second, based on this projection, one needs to design a memory system resource shifting algorithm that moves an appropriate amount of memory system resources from the GPU to the CPU cores so that the GPU continues to perform just around the target QoS threshold. These two algorithms form the crux of our proposal. In the rest of the study, we consider 40 FPS to be the target QoS threshold for 3D scene rendering leaving a 10 FPS cushion for handling any momentary dip.

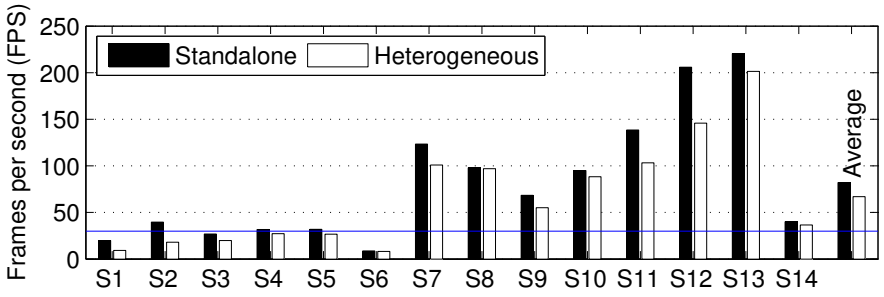


Figure 4.2: Comparison of GPU frame rate in standalone and heterogeneous execution. The reference line shows 30 FPS mark.

Two primary memory system resources that are shared between the GPU and the CPU cores are LLC capacity and DRAM bandwidth. Therefore, for best outcome, an algorithm that dynamically shifts memory system resources from the GPU to the CPU applications based on estimated performance levels would try to focus on both LLC capacity and DRAM bandwidth. Prior studies have explored bypassing the LLC for GPU read misses thereby targeting only the LLC capacity [71]. An ideal LLC bypass algorithm for GPU applications would only free up portion of the

LLC for CPU applications while leaving the DRAM bandwidth consumption of the GPU unchanged. However, since designing an ideal bypass algorithm is difficult, it is expected that the DRAM bandwidth consumption of the GPU will increase when LLC bypassing for GPU read misses is enabled. Since the GPU is designed to have high latency-tolerance, these additional LLC misses may not hurt the GPU performance. However, the extra DRAM bandwidth consumed by these additional GPU LLC misses can lead to significant drop in CPU performance.

Figure 4.3 shows the impact on CPU workload performance when all GPU read misses are forced to bypass the LLC. On average, compared to the heterogeneous mode of execution without LLC bypass for GPU read misses, the CPU applications lose 2% performance. While there are CPU applications that gain as much as 10% (S4), there are also applications that lose as much as 14% (S9).[1] The CPU applications which fail to utilize the additional LLC capacity created through GPU read miss bypass start suffering due to increased DRAM bandwidth contention. The GPU applications enjoy a significant volume of reuses from the LLC in the baseline. When all GPU fills bypass the LLC, the GPU applications lose these reuses and significantly increase the DRAM traffic. This increased DRAM bandwidth pressure hurts the performance of both CPU and GPU. We revisit this aspect in Section 4.4 when we evaluate HeLM, the state-of-the-art LLC management policy that relies on selective LLC bypass of GPU fills [71]. In summary, any algorithm that dynamically shifts memory system resources from the GPU to the CPU applications must be able

[1] Figure 3.3 in Chapter 3 showed the performance results when 100% GPU fills are forced to bypass the LLC. However, those results also included the heterogeneous mixes containing the GPGPU applications. Bypassing GPU fills in GPGPU applications brings significant improvement in the performance of the co-scheduled CPU workload mixes. As a result, averaged across all mixes, the CPU improved in performance by 5% when all GPU fills bypass the LLC in the 1C1G configuration. In this chapter, we focus on only the subset of the heterogeneous mixes that contain 3D scene rendering workloads.

to create LLC capacity *as well as* DRAM bandwidth for the CPU applications. In our proposal, we use the important insight that throttling the GPU access rate to the LLC can achieve both the goals. A slowed down GPU access rate automatically ages the GPU blocks faster in the LLC leading to their eviction from the LLC and creating more LLC capacity for the CPU applications. Also, a slowed down GPU access rate to the LLC naturally lowers the GPU's demand on the DRAM bandwidth; even though the volume of GPU LLC misses increases, these misses are sent to the DRAM at a dynamically controlled rate that is just enough for the GPU application to meet the target QoS threshold.

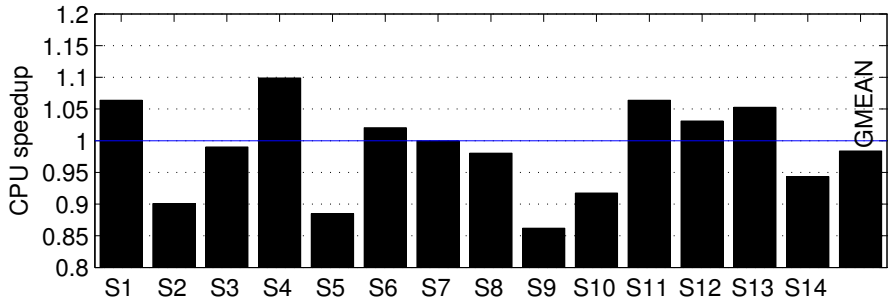


Figure 4.3: CPU speedup when all GPU read miss fills are forced to bypass the LLC.

4.2 Memory Access Management

In this section, we present our proposal on memory access management for CPU-GPU heterogeneous processors. Our proposal involves a three-step algorithm. In the first step, we dynamically estimate the frame rate of the GPU application. In the second step, depending on the estimated frame rate, we determine an appropriate rate at which GPU accesses are sent to the LLC. In the third step, depending on the estimated frame rate, the CPU access priority is altered in the DRAM access

scheduler. If the estimated frame rate falls below the target QoS threshold, the second and the third steps are not invoked and the GPU application continues to run in the baseline heterogeneous mode without any change in the LLC access rate. The predictive model for online frame rate estimation used in the first step is discussed in Section 4.2.1. The access throttling mechanism is discussed in Section 4.2.2. The changes in the DRAM access scheduler are discussed in Section 4.2.3. Section 4.2.4 quantifies the storage overhead of our proposal.

4.2.1 Dynamic Frame Rate Computation

For the 3D scene rendering workloads, our proposal requires knowledge of the frame rate ahead of the actual completion of the frame so that the GPU access rate and the DRAM access scheduler can be adapted accordingly. Such dynamic prediction of the frame rate requires answering the following two questions.

1. How much is the amount of work per frame?

2. Given the rendering speed and the amount of work per frame, how to predict the frame rate?

To answer these questions, we divide the entire rendering process into two phases, namely, learning phase and prediction phase. In the learning phase, we monitor rendering of a frame and measure the amount of work done and the time it takes to complete. Once this information is obtained for one complete frame, we switch to the prediction phase. In this phase, the data collected in the learning phase is used to predict the frame rate. To ensure that the observed data follows the collected data, new observations are cross-verified against the learned data. If it is found that the observed values differ from the learned values by more than a threshold amount,

the learned data is discarded and we switch back to the learning phase. Figure 4.4 demonstrates a sequence of phase transitions that happen in a hypothetical rendering job. Rendering begins in the learning phase and continues to remain in that phase till point A, where it is determined that the data for one complete frame has been collected. Thus, at this point we transition into the prediction phase. Now, rendering continues in this phase up to point B, where it is found that the learned data is no more valid and thus, we transition back to the learning phase to collect fresh data. At point C, we again transition to the prediction phase. This cycle is repeated until the entire rendering job completes. Since we would like to maximize the number of frames in the prediction phase for having good prediction coverage, the success of this scheme improves if the amount of work in each frame within a set of consecutive frames remains more or less constant.

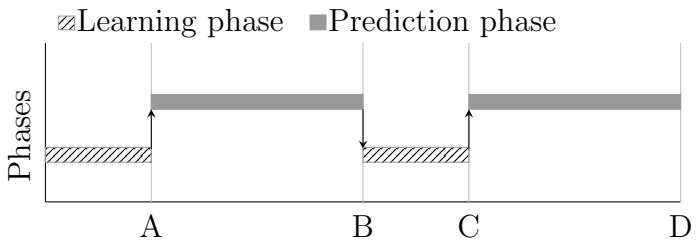


Figure 4.4: Rendering phases

4.2.1.1 Learning Phase

Rendering of a frame in a rudimentary sense boils down to computing the color values of all pixels from the input geometry and updating these values into a buffer commonly known as the render target (RT). In general, a single pixel in the RT can get overdrawn multiple times depending on the order in which the geometry primitives arrive for rendering and their depth. This complicates the estimation of

the amount of work involved in rendering a frame. We divide the RT into equal sized $t \times t$ render target tiles (RTT). We divide the entire rendering of a frame into render target planes (RTP). Each RTP represents a batch of updates that cover all tiles of the RT. Therefore, the number of RTPs is the number of updates that cover all tiles of the RT. This arrangement is shown in Figure 4.5.

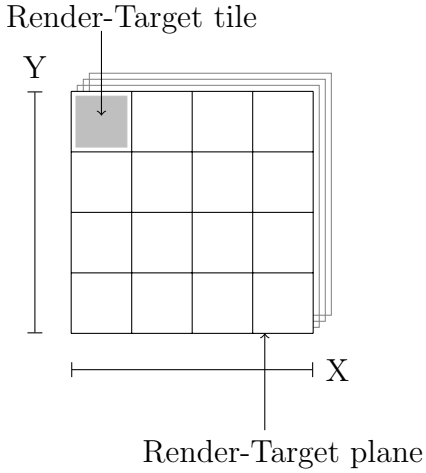


Figure 4.5: Render-target plane and tile

We maintain a 64-entry RTP information (RTPi) table in the GPU. For a frame, each entry of this table maintains a valid bit and four pieces of information about a distinct RTP. These four pieces of information are: (i) total number of updates to the RTP, (ii) the number of cycles to finish the RTP, (iii) the number of RTTs in the RTP, and (iv) the number of shared LLC accesses (i.e., GPU render cache misses) made by the GPU for the entire RTP. Our implementation assumes each of the four fields to be four bytes in size. The first three fields are used in the prediction phase. The LLC access count is passed on to the access throttling algorithm (see Section 4.2.2) for computing the maximum throttling rate. If the number of RTPs in a frame exceeds 64, the last entry of the table is used to accumulate the data for all subsequent RTPs.

4.2.1.2 Prediction Phase

Our frame rate prediction model uses the RTP count and cycles per RTP recorded during the learning phase to predict the current number of cycles per frame. If the number of RTPs in a frame i is N^i_{rtp} and the average number of cycles per RTP is C^i_{rtp}, then the number of estimated cycles F_i required to render frame i is given by Equation 4.1.

$$F_i = C^i_{rtp} \times N^i_{rtp} \tag{4.1}$$

Although N^i_{rtp} is obtained directly from the data collected during the learning phase, C^i_{rtp} for the frame being rendered currently has to be extrapolated using the number of cycles the frame has taken so far and the cycles recorded in the RTP table, so that the current rendering speed of the frame can be taken into account for obtaining the full frame cycle count. Suppose the fraction of a frame that has been rendered so far is λ, the average number of cycles per RTP seen in the current frame is C^i_{inter}, and the average number of cycles per RTP recorded during the learning phase is C^i_{avg}. Therefore, C^i_{rtp} can be computed using Equation 4.2.

$$C^i_{rtp} = \lambda \times C^i_{inter} + (1 - \lambda) \times C^i_{avg} \tag{4.2}$$

If we substitute Equation 4.2 into Equation 4.1, we obtain the final expression for the predicted number of cycles per frame as shown in Equation 4.3.

$$F_i = (\lambda \times C^i_{inter} + (1 - \lambda) \times C^i_{avg}) \times N^i_{rtp} \tag{4.3}$$

We note that λ is computed as the ratio of the number of RTPs completed so far in the frame being currently rendered to the total number of RTPs observed during

the learning phase.

4.2.2 Access Throttling Mechanism

In this section, we discuss our GPU access throttling mechanism. This mechanism is invoked only if the GPU is found to be meeting a target frame rate. For the 3D scene rendering workloads that are predicted to meet a target frame rate, the rate with which the GPU can send access requests to the shared LLC is throttled down. Throttling GPU accesses before the LLC has two implications. First, the GPU blocks in the LLC are accessed less frequently causing them to age faster compared to the CPU blocks. This replaces the GPU blocks early increasing the number of GPU misses and increasing the average residency of the CPU blocks in the LLC. Second, the GPU accesses that miss in the LLC are seen by the DRAM at a slower rate automatically shifting a bigger proportion of the DRAM bandwidth to the CPU workloads. Our access throttling proposal allows N_G GPU accesses within a window of W_G GPU cycles, thereby enforcing an average GPU access rate of N_G/W_G. This is implemented as follows.

Every GPU access must go through a translation from the GPU address to the global physical address before the access can be routed to the correct LLC bank. This translation is accomplished by looking up the graphics translation table (GTT) resident in the GPU (please refer to Chapter 2 for detail). At the beginning of a window, N_G and W_G are initialized. W_G is decremented on every GPU cycle and N_G is decremented on every GPU access to the GTT. As soon as N_G reaches zero, the GTT ports are disabled until W_G reaches zero. As a result, during this period the GPU is denied access to the LLC. When the GPU requests are denied access

to the LLC, they are held back inside the GPU and occupy GPU resources such as request buffers and MSHRs attached to the caches internal to the GPU. This resource contention is modeled in detail in our evaluation. Any negative influence that this resource contention may have on performance automatically gets reflected in the progress of rendering and is captured by our frame rate estimation algorithm. The estimation, in turn, feeds back into the throttling mechanism.

Choosing Values for W_G and N_G. We need an algorithm that automatically adjusts N_G and W_G based on the estimated and the target frame rates. Let the number of cycles per frame at the current predicted frame rate be C_P, the number of cycles per frame at the target frame rate be C_T, and the number of LLC accesses per frame be A (this is recorded during the learning phase of the frame rate prediction model as discussed in Section 4.2.1.1). Therefore, if $C_P < C_T$ meaning that the GPU is delivering better frame rate than the target, we would like to increase W_G by $(C_T - C_P)/A$ while holding N_G at one. This increment is done gradually at a step of two in each window. On the other hand, if $C_P \geq C_T$, access throttling is disabled by setting W_G to zero and N_G to one. Small oscillation around the target frame rate can be avoided by disabling throttling within a small guard-band around the target frame rate. Figure 4.6 shows the flow of the throttling mechanism.

4.2.3 DRAM Access Scheduler

The goal of our DRAM scheduling policy is to shift bandwidth to the CPU if the GPU is able to meet target QoS. This is implemented using a simple scheme. If the GPU is currently predicted to meet the target frame rate, the CPU requests are prioritized over the GPU requests. Within a bank, among the requests that can

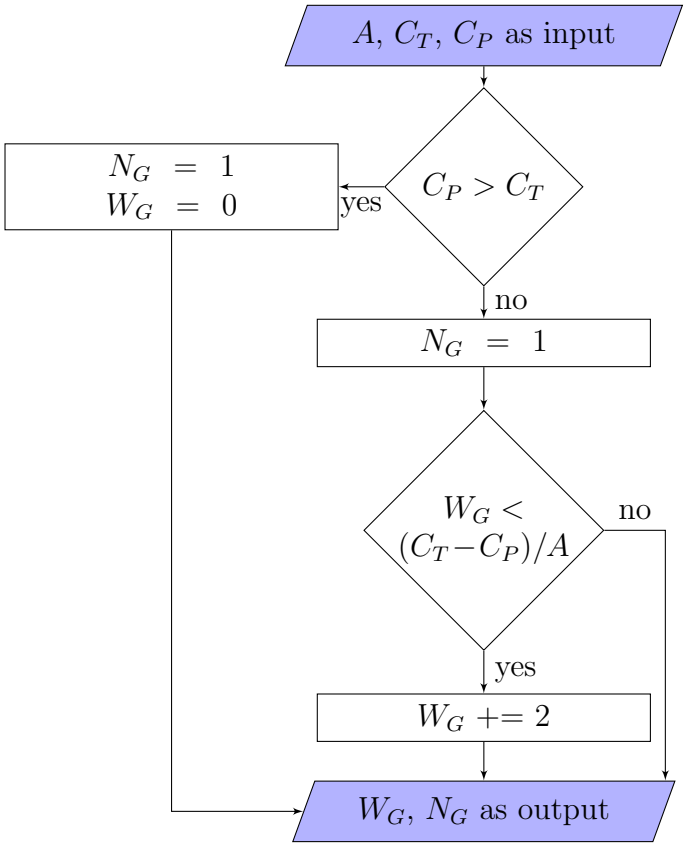


Figure 4.6: Flow of the algorithm that throttles LLC accesses from the GPU. A, C_T, and C_P are input parameters. W_G and N_G are outputs.

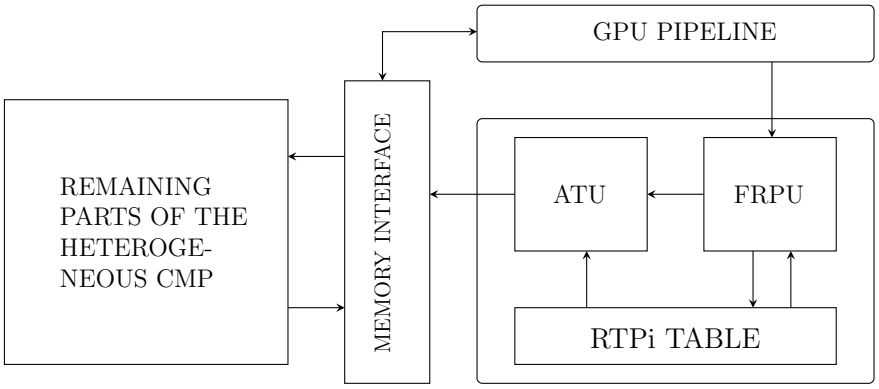


Figure 4.7: Architecture of the frame rate prediction and access throttling mechanism. FRPU is the frame rate prediction unit implementing the prediction mechanism discussed in Section 4.2.1. ATU is the access throttling unit implementing the algorithm discussed in Section 4.2.2. RTPi table was introduced in Section 4.2.1.1.

enjoy row buffer hits, the scheduling policy first schedules the CPU accesses in FCFS order and then considers the rest. When a new row needs to be activated in a bank, the oldest CPU access is given priority over the global oldest access. The scheduler follows the baseline FR-FCFS algorithm if the GPU fails to meet the target frame rate. Figure 4.7 summarizes our entire proposal.

4.2.4 Storage Overhead

The storage overhead of our proposal is small. It involves the RTPi table having 64 entries, each entry being 129 bits leading to a total investment of just over 1 KB. Our proposal also requires two short registers to maintain W_G and N_G. One state bit is required to indicate whether the DRAM access scheduler should invoke the baseline policy or the policy with enhanced CPU priority. The primary storage overhead arises from the RTPi table and it is important to note that the accesses to this table are not on the critical path of the GPU accesses. Updates to this table happen

off the critical path and this table is read only periodically at a certain interval for computing the projected frame rate.

4.3 Related Work

Dynamic frame rate estimation has been studied in the context of tile-based deferred rendering (TBDR) [37], commonly found in mobile GPUs [80, 86]. Dynamic progress estimation of GPUs has also been explored in the presence of prior profile information such as the number of memory accesses issued by the GPU application [103]. In contrast, our proposal does not make any assumption about the implementation of the rendering pipeline and nor does it require a profile pass.

Request throttling to reduce unfairness in the memory system of CPU-based CMPs has been explored [18]. The mechanism and goal of our proposed GPU access throttling are, however, entirely different. Also, in the context of GPGPU applications, there have been studies to throttle up/down the number of active warps and active thread blocks in the shader cores based on memory system congestion and GPU idle cycles [45, 46]. These are primarily shader core-centric proposals and are not effective for the 3D scene rendering workloads that generate a large volume of memory accesses from the fixed function units such as the texture samplers, color blenders, depth test units, etc..

To gain better understanding of this class of shader core-centric throttling mechanisms, we discuss the shortcomings of the balanced concurrency management (CM-BAL) proposal [46] in more detail. CM-BAL scales up or down the maximum number of ready shader threads based on the average stalls observed with different thread configurations. We observe that CM-BAL fails to adequately throttle

the GPU frame rate for three reasons. First, throttling only the shader threads primarily impacts the texture access rate to the LLC because the texture samplers are directly attached to the shader cores and the texture accesses are triggered by texture filtering instructions of the shader program. However, in the 3D scene rendering workloads considered by us, the texture accesses, on average, constitute only 25% of all LLC accesses coming from the GPU. As a result, throttling only the texture access rate is not enough. The render output pipeline (ROP) consisting of the color blenders, depth test units, and color writers receives shaded and textured fragments from the shader cores and is responsible for writing out the final pixel colors to the render buffer. It is necessary to drive the usually over-utilized ROP to an under-utilized region of operation to be able to see an effect of shader core throttling on the LLC access rate from the ROP. In practice, this is impossible to achieve through shader core throttling alone. While it is feasible to throttle individual units of the 3D rendering pipeline at appropriate rates, this leads to a design that is far more complex than what we propose. Our proposal does not focus on any particular unit in the rendering pipeline; it throttles the collective rate at which the GPU can access the LLC. Second, different applications show different performance sensitivity toward texture access rate, which experiences the first order impact of shader core throttling. So, throttling the texture access rate is not guaranteed to have a significant performance influence on the GPU. Third, at run-time when the CM-BAL policy is applied, only a fraction of the texture accesses undergo throttling further diminishing the overall performance impact.

DRAM access scheduling has been explored for CPU-based platforms, discrete GPU parts, and heterogeneous CMPs. In the following, we discuss the contributions relevant to the discrete GPU parts and heterogeneous CMPs. The memory access

scheduling studies for the discrete GPU parts have been done with the GPGPU applications. These studies have explored memory access scheduling to minimize the latency variance among the threads within a warp [6], to accelerate the critical shader cores that do not have enough short-latency warps which could hide long memory latencies [41], and to minimize a potential function so that an appropriate mix of shortest-job-first and FR-FCFS can be selected with the overall goal of accelerating the less latency-tolerant shader cores [60]. There have been studies on warp and thread block schedulers for improving the memory system performance [2, 39, 40, 45, 61, 63].

Motivated by the bandwidth-sensitive nature of the massively threaded GPU workloads and the deadline-bound nature of the 3D scene rendering workloads executed on the GPUs, prior proposals have explored specialized memory access schedulers for heterogeneous systems [3, 37, 78, 95]. The staged memory scheduler (SMS) clubs the memory requests from each source (CPU or GPU) into source-specific batches based on DRAM row locality [3]. Each batch is next scheduled with a probabilistic mix of shortest-batch-first (favoring latency-sensitive CPU jobs) and round-robin (enforcing fairness among bandwidth-sensitive CPU and GPU jobs). The dynamic priority (DynPrio) scheduler [37], proposed for mobile heterogeneous platforms, employs dynamic progress estimation of tile-based deferred rendering (TBDR) and offers the GPU accesses equal priority as the CPU accesses if the GPU progress lags behind the target frame rendering time. Also, during the last 10% of the left time to render a frame, the GPU accesses are given higher priority than the CPU accesses. The progress estimation algorithm used by DynPrio is designed specifically for the GPUs employing TBDR, typically supported only in mobile GPUs such as ARM Mali [80], Kyro, Kyro II, and PowerVR from Imagination

Technologies, and Imageon 2380, Xenos, Z430, and Z460 from AMD [86]. The DynPrio scheduler study, however, shows the inefficiency of a previously proposed static priority scheduler that always offers higher priority to the CPU accesses [95]. The option of statically partitioning the physical address space between the CPU and GPU datasets and assigning two independent memory controllers to handle accesses to the two datasets has been explored [78]. A subsequent study has shown that such static partitioning of memory resources can lead to sub-optimal performance in heterogeneous systems [46]. In Section 4.4, we present a quantitative comparison of our proposal with DynPrio and two variants of SMS.

There have been studies on managing the shared LLC in a CPU-GPU heterogeneous processor. Two of these studies (thread-aware policy [62] and dynamic reuse probability-aware policy [85]) propose new insertion, promotion, and replacement policies for LLC blocks. Another study (HeLM) proposes to selectively bypass the LLC for GPU read misses coming from the shader cores that are dynamically identified to be latency-tolerant, thereby opportunistically shifting LLC capacity to the co-executing CPU workloads [71]. In Section 4.1, we have already pointed out the shortcomings of such a mechanism that is based purely on LLC bypass techniques. Nonetheless, since our proposal is closer to HeLM in its goal, we present a quantitative comparison of our proposal with HeLM in Section 4.4.

4.4 Simulation Results

In this section, we evaluate our proposal on a simulated heterogeneous CMP with four CPU cores and one GPU. With each GPU workload, we co-execute a mix of four CPU applications. We divide the discussion into evaluation of the individual

components that constitute our proposal.

Accuracy of Dynamic Frame Rate Estimation. Figure 4.8 shows the percent error observed in our dynamic frame rate estimation technique. A positive error means over-estimation and a negative error means under-estimation. Several applications have zero error. Among the applications that have non-zero error, the maximum over-estimation error is 6% (UT2004) and the maximum under-estimation error is 4% (COR). The average error across all applications is less than 1%.

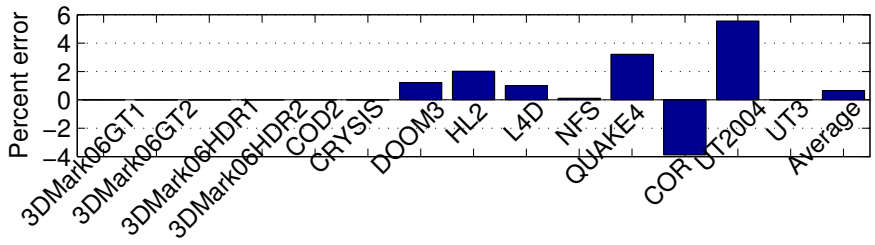


Figure 4.8: Percent error in dynamic frame rate estimation.

Evaluation of Access Throttling. Figure 4.9 quantifies the performance of our GPU access throttling mechanism. In this evaluation, we set the target frame rate as 40 FPS for the GPU applications. Referring to the last column of Table 2.2, we see that there are six applications that have frame rates higher than this target. The rest of the applications never go above 40 FPS for the selected frame sequence. Therefore, our proposal will be able to apply access throttling to these six applications. The left panel of Figure 4.9 quantifies average FPS for the GPU applications. For each application, we show three bars. The leftmost bar corresponds to baseline. The middle bar corresponds to a system with access throttling enabled. The rightmost bar corresponds to a system with access throttling enabled and the CPU applications given higher priority over the GPU in the DRAM access scheduler. The right panel of this figure shows the weighted speedup (normalized

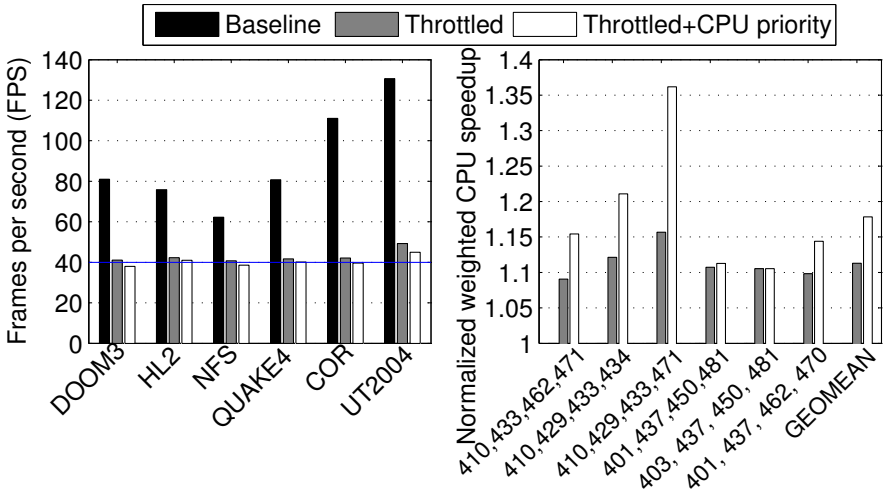


Figure 4.9: Left: FPS of GPU applications that are amenable to access throttling. Right: weighted CPU speedup for the mixes when the GPU application in the mix is throttled. The CPU application mixes are shown in terms of the combination of the SPEC application ids along the x-axis of the right panel.

to the baseline, which is at 1.0) achieved by the four-way multi-programmed CPU workloads when the corresponding GPU workload in the mix is undergoing access throttling. We identify each CPU application in a mix by its SPEC id. The GPU application results confirm that the six applications operate just around 40 FPS when access throttling is enabled. While this represents the average frame rate over the multi-frame sequence for each application, we also verified that each frame within the sequence meets the target frame rate. For the CPU applications, the mixes improve significantly offering an average 11% speedup with GPU access throttling alone; the average speedup improves to 18% when higher CPU priority is enabled in the DRAM access scheduler.

To further understand the sources of CPU performance improvement, Figure 4.10 quantifies the LLC miss count of the GPU applications (left panel) and the corresponding CPU workload mixes (right panel) normalized to the baseline. The GPU applications suffer from an average 39% increase in LLC miss count when GPU ac-

cess throttling is enabled. This number further increases to 42% when CPU priority in the DRAM access scheduler is boosted in addition to GPU access throttling. As already explained, this is an expected behavior resulting from faster aging of the GPU blocks in the LLC due to lowered LLC access rate of the GPU application. In addition to GPU access throttling, when the CPU priority in the DRAM access scheduler is boosted, the CPU fills return faster to the LLC, thereby evicting the aged GPU blocks even more quickly. The right panel shows that the additional LLC space created due to this leads to a 4% and 4.5% average reduction in CPU LLC miss counts for the throttled and throttled+CPUpriority configurations, respectively.

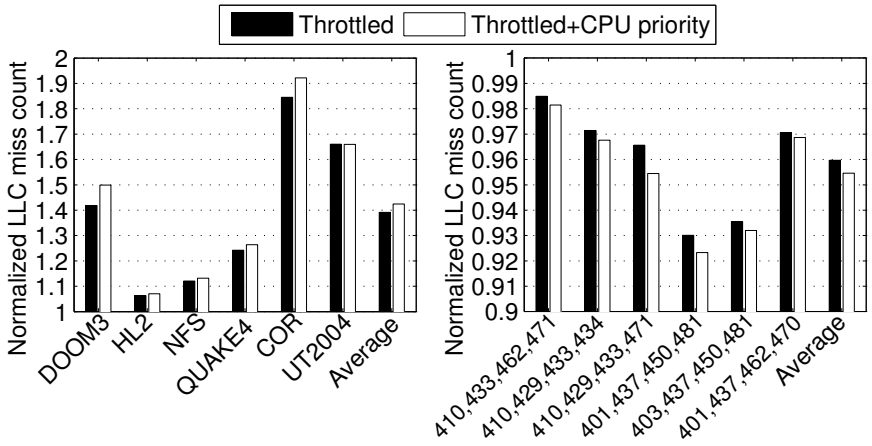


Figure 4.10: Left: normalized LLC miss count of GPU applications that are amenable to access throttling. Right: normalized LLC miss count of CPU workloads when the GPU application in the mix is throttled.

The significant increase in the LLC miss count of the GPU applications can be of concern because this may lead to higher DRAM bandwidth consumption defeating the very purpose of GPU access throttling. However, it is important to note that these misses occur over a much longer period of time due to a lowered frame rate. Our access throttling algorithm automatically adjusts the throttling rate taking all these into consideration so that the frame rate hovers close to the target level. Fig-

ure 4.11 substantiates this fact by quantifying the average DRAM bandwidth (read and write separately shown) consumed by the GPU applications normalized to the baseline. [1] On average, the GPU bandwidth demand reduces by 35% and 37%

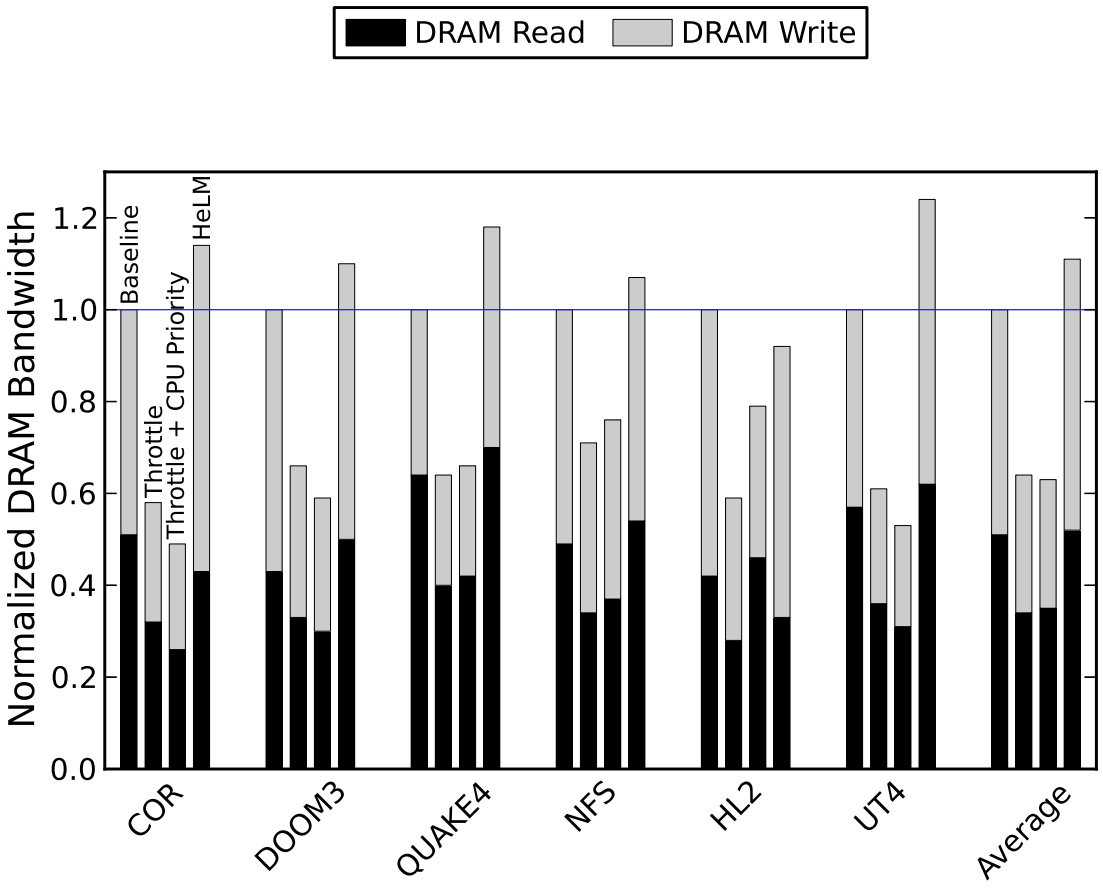


Figure 4.11: Normalized DRAM bandwidth (read and write) consumed by GPU applications that are amenable to access throttling.

for the throttled and throttled+CPUpriority configurations, respectively. Both read and write bandwidth demands go down by significant amounts, across the board, where as, for HeLM due to read miss bypass DRAM bandwidth increases signifi-

[1] In a few GPU applications (e.g., DOOM3 and HL2), the volume of writes can be more than the volume of reads because the rendering pipeline can create fully dirty color or depth lines in the internal ROP caches and later flush them out to the LLC for allocation without doing a DRAM read.

cantly. In summary, our proposal frees up more than one-third GPU bandwidth for the CPU workloads to consume. A comparison with the right panel of Figure 4.10 reveals that DRAM bandwidth shifting is a bigger benefit of our proposal than LLC miss saving for CPU workloads.

Comparison to Related Proposals. Several DRAM scheduling policies have been proposed for heterogeneous CMPs and evaluated on mixes of 3D scene rendering workloads and CPU workloads. In the following, we compare our proposal against staged memory scheduling (SMS) [3] and dynamic priority scheduler (DynPrio) [37]. Additionally, we present a comparison with HeLM, the state-of-the-art LLC management policy for CPU-GPU heterogeneous processors [71]. We evaluate two versions of SMS, namely, one with a probability of 0.9 of using shortest-job-first (SMS-0.9) and the other with this probability zero i.e., it always selects a round-robin policy (SMS-0). SMS-0.9 is expected to favor latency-sensitive CPU jobs while SMS-0 is expected to favor GPU jobs. DynPrio makes use of our frame rate estimation technique to compute the time left in a frame. Figure 4.12 compares the proposals for the heterogeneous mixes containing GPU applications that meet the target 40 FPS. The upper panel shows that all proposals deliver higher than 40 FPS. Our proposal (ThrotCPUprio) opportunistically applies GPU access throttling and CPU prioritization in the DRAM scheduler to deliver an FPS that is just around the target. As a result, our proposal is able to improve the CPU mixes most (lower panel of Figure 4.12). On average, SMS-0.9, SMS-0, DynPrio, HeLM, and our proposal improve the performance of the CPU mixes by 4%, 4%, 10%, 3%, and 18%, respectively. The performance improvement achieved by HeLM is low because it suffers from an increased DRAM bandwidth consumption arising from the additional GPU misses that result from aggressive LLC bypass of GPU

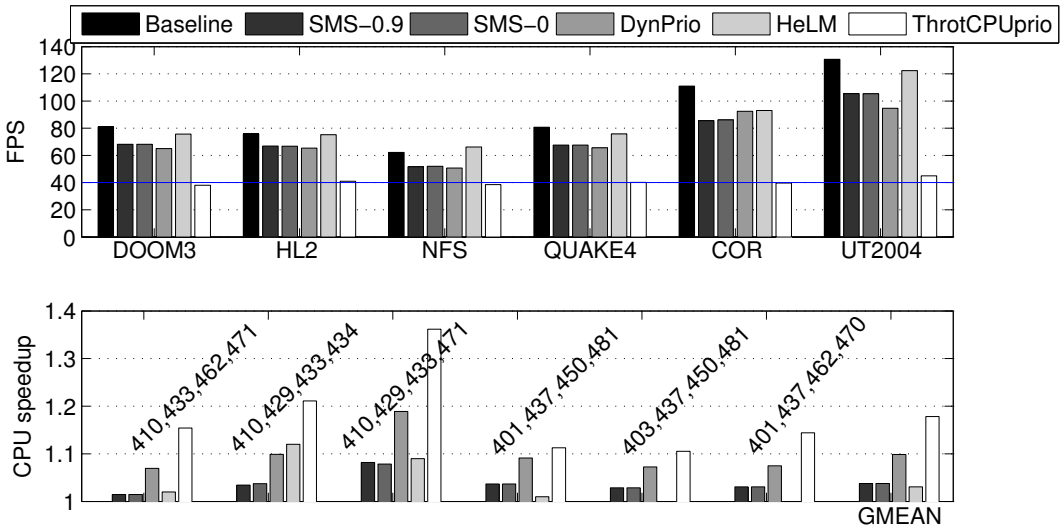


Figure 4.12: FPS of GPU applications (top panel) and normalized weighted CPU speedup (bottom panel) for the mixes with high frame rate GPU applications.

fills. Our proposal remains disabled in the remaining mixes containing the GPU

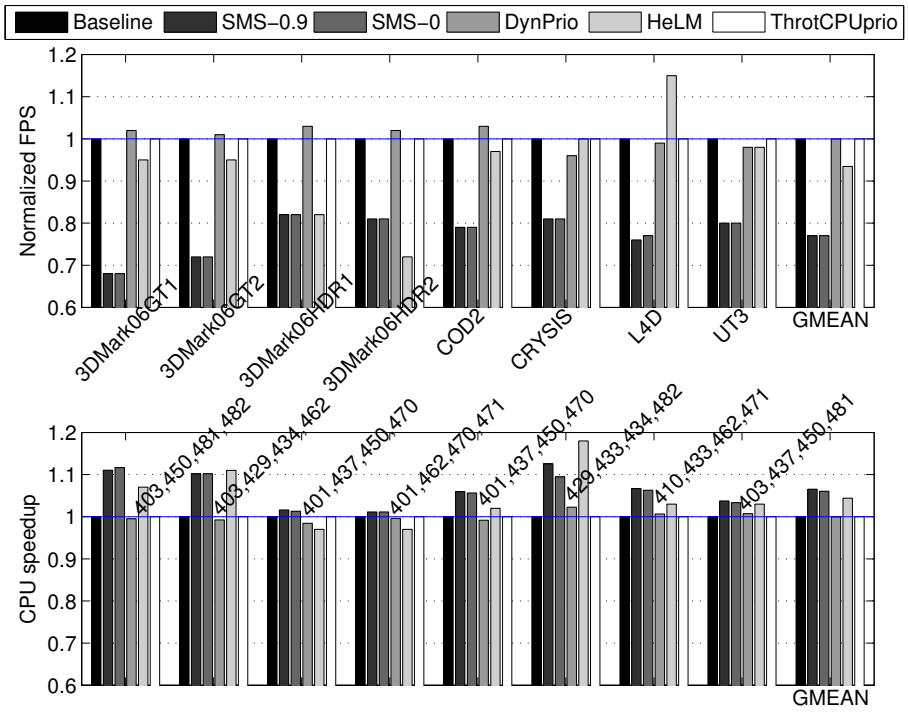


Figure 4.13: FPS speedup (top panel) and weighted CPU speedup (bottom panel) over the baseline for the mixes with low frame rate GPU applications.

applications that fail to meet the target FPS. For completeness, Figure 4.13 shows

the comparison for these mixes. The upper panel evaluates the proposals for the GPU applications. SMS suffers from large losses in FPS due to the delay in batch formation. DynPrio fails to observe any overall benefit because it offers express bandwidth to the GPU application only during the last 10% of a frame time and otherwise the CPU and GPU are offered equal priority as in the baseline FR-FCFS. HeLM suffers from an average loss of 7% in FPS due to DRAM bandwidth shortage resulting from the additional GPU misses that arise due to aggressive LLC bypass.

SMS-0.9 and SMS-0 improve CPU mix performance (lower panel of Figure 4.13) by 7% and 6%, respectively, while suffering large losses in GPU performance. DynPrio delivers the same level of performance as the baseline for both GPU and CPU workloads. HeLM enjoys a 4% average improvement in CPU mix performance. In summary, these results clearly indicate that the GPU performance can be traded off to improve CPU performance and vice-versa in such heterogeneous platforms. To understand the overall performance in such scenarios, we assign equal weightage to the CPU and GPU performance and derive the overall performance of the heterogeneous processor. Figure 4.14 shows these results for the mixes containing the GPU applications that fail to meet the target FPS. [1] On average, our proposal and Dyn-

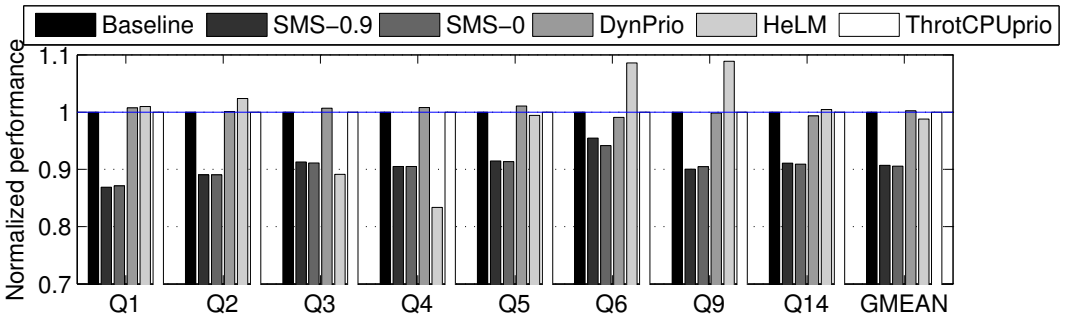


Figure 4.14: Normalized performance of the heterogeneous processor for the mixes with low frame rate GPU applications when the CPU and GPU are given equal weight in terms of performance.

Prio both deliver baseline performance for these mixes, while both variants of SMS suffer from large losses. HeLM performs 1% worse than the baseline on average.

Exploring the Option of Phased Execution. Our proposal uniformly throttles the GPU access rate throughout the execution. Another option is to divide the execution into two types of coarse grain phases that alternate. The execution begins with a CPU+GPU (heterogeneous) phase where the GPU completes rendering of one frame. If the GPU application is able to meet the target frame rate, the execution transitions to a CPU-only phase. In this phase, rendering of subsequent frames is suspended until the desired number of cycles has elapsed so that the GPU is able to perform just at the target frame rate. All memory system resources can be utilized by the CPU in this phase. The execution keeps alternating between the heterogeneous and CPU-only phases. For example, if the target frame rate is 40 FPS and the GPU completes rendering of a frame in the heterogeneous mode in 0.02 seconds, it will suspend rendering of the next frame for the next 0.005 seconds. During these 0.005 seconds, the CPU can enjoy exclusive ownership of all memory system resources. Figure 4.15 shows the performance improvement of the CPU workload mixes in our proposal normalized to the option of phased execution. We identify the CPU workloads on the x-axis by the associated GPU application name in the heterogeneous mix. Our proposal improves CPU performance by 3% on average and up to 6% compared to the option of phased execution. The CPU workload associated with COR is the only case that gains slightly more with phased execution.

[1] For the mixes where the GPU application already meets the target FPS, this kind of a combined CPU-GPU performance metric is irrelevant because the GPU performance goal is already satisfied and an evaluation of the CPU performance improvement, as shown in Figure 4.12, is sufficient.

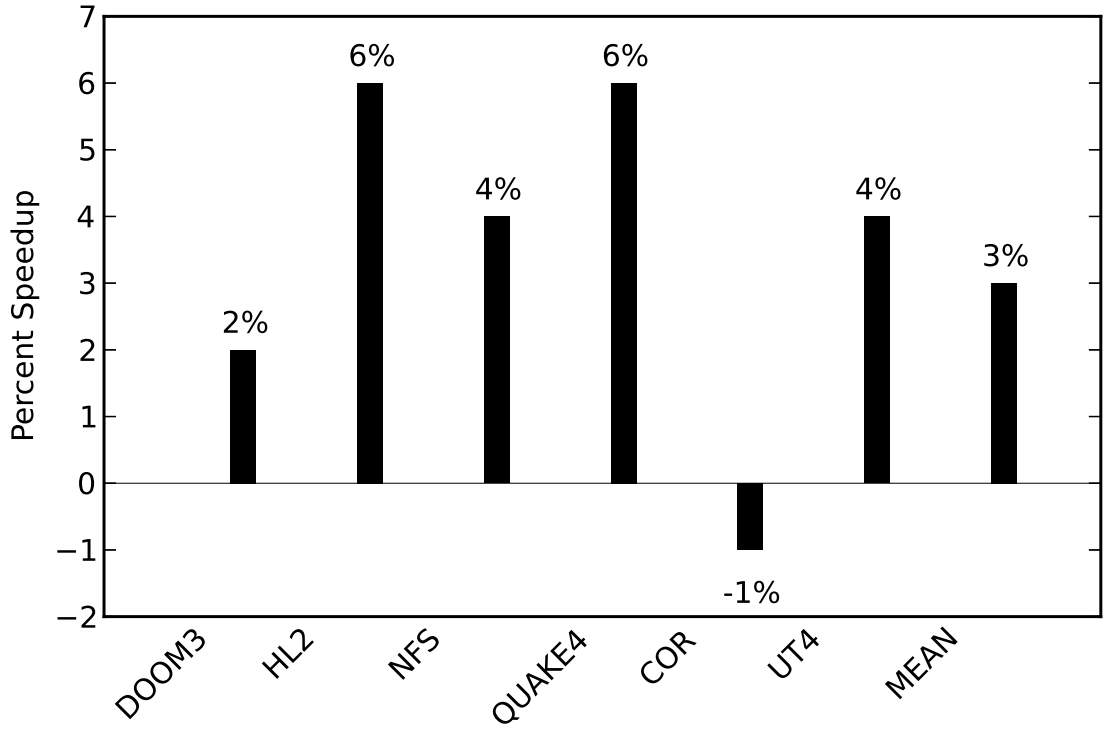


Figure 4.15: Performance of CPU application mixes in our proposal normalized to the option of phased execution (higher is better). We use the GPU application names on the x-axis to uniquely identify the associated CPU mix.

The reason for our proposal to be better in the other cases is the reduction in cache and DRAM contention throughout the execution. The phased execution alternative fails to recover the performance lost by the CPU during the heterogeneous phase of execution as efficiently as our proposal does. This is primarily because the CPU-only phase of the CPU workload mix may not be equally memory-intensive as the heterogeneous phase. As a result, the CPU-only phase may not be able to fully exploit the resources temporarily freed up due to suspension of the GPU application. However, we find that for the CPU workloads that are uniformly memory-bound throughout the execution, phased execution performs better than our proposal. For such workloads, running both CPU and GPU together for the entire execution leads

to higher overall DRAM contention than observed in phased execution.

4.5 Conclusions

We have presented a novel memory access management mechanism for heterogeneous CMPs. The proposed mechanism dynamically shifts LLC capacity and DRAM bandwidth to CPU applications from the co-executing GPU application whenever the GPU application meets the target frame rate. Two light-weight, yet highly effective, algorithms form the crux of our proposal. The first algorithm accurately estimates the projected frame rate of a GPU application. Based on this estimation, the second algorithm computes the effective GPU access rate to the LLC and assists the DRAM access scheduler to decide if CPU priority should be boosted. Detailed simulation studies show that our proposal achieves its goal of offering a bigger share of the memory system resources to the CPU when the GPU does not need it. For the heterogeneous mixes containing GPU applications that meet the target frame rate, our proposal improves the CPU performance by 18% on average while requiring just over a kilobyte of additional storage.

Chapter 5

GPU Criticality-driven Memory Management

This chapter presents our proposal on memory access scheduling driven by criticality of GPU accesses for CPU-GPU heterogeneous system. Different GPU access streams originating from different parts of the GPU rendering pipeline behave very differently compared to the typical CPU pipeline requiring new techniques for GPU access criticality estimation. We propose a novel queuing network model to estimate the performance-criticality of the GPU access streams. If a GPU application performs below the quality of service requirement (e.g., frame rate in 3D rendering), the memory access scheduler uses the estimated criticality information to accelerate the critical GPU accesses. Detailed simulations done on a heterogeneous chip-multiprocessor model with one GPU and four CPU cores running DirectX, OpenGL, and CPU application mixes show that our proposal improves the GPU performance by 15% on average without degrading the CPU performance much. Extensions proposed for the mixes containing GPGPU applications, which do not have any quality

of service requirement, improve the performance by 7% on average.

The rest of the chapter is organized as follows. Section 5.1 presents a motivational study demonstrating varying sensitivity of different GPU streams toward rendering performance. Section 5.2 presents different components of our proposal followed by related work, experimental evaluation, and conclusions in Sections 5.3, 5.4, and 5.5, respectively.

5.1 Motivation

Different types of data are accessed by the programmable and the fixed function units in a GPU rendering pipeline. Examples of such data include vertex data, vertex index data, pixel color data, texture sampler data, pixel depth data, hierarchical depth data [24], shader cores' instruction and data, blitter data, etc.. An access from a data stream looks up the internal cache hierarchy of the GPU dedicated to that stream and, on a miss, looks up the LLC shared between the GPU and CPU cores. The LLC misses are served by the DRAM. In this section, we demonstrate that the sensitivity of different types of GPU access streams toward memory system optimization is not uniform necessitating a stream-wise criticality measure.

5.1.1 GPU Stream-wise Criticality

Figure 5.1 shows the distribution of DRAM read accesses across different stream types coming from the GPU for fourteen DirectX and OpenGL workloads. Each workload renders a multi-frame segment of a popular PC game. These data are collected on a simulated heterogeneous CMP. We consider the following stream categories: color (C), texture sampler (T), depth (Z), blitter (B), and everything

else clubbed into the "other" (O) category.[1] Figure 5.1 shows that, in general, the color, texture, and depth streams constitute the larger share of the DRAM accesses from the GPU; the actual distribution varies widely across applications.

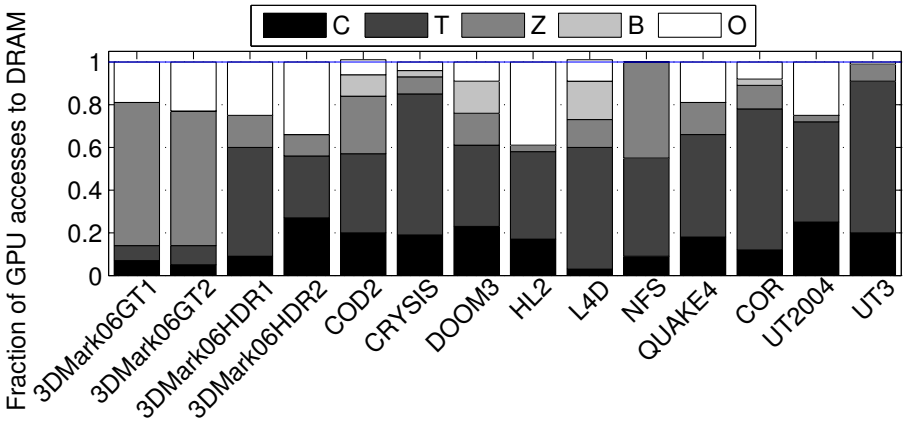


Figure 5.1: Distribution of DRAM accesses from 3D scene rendering workloads.

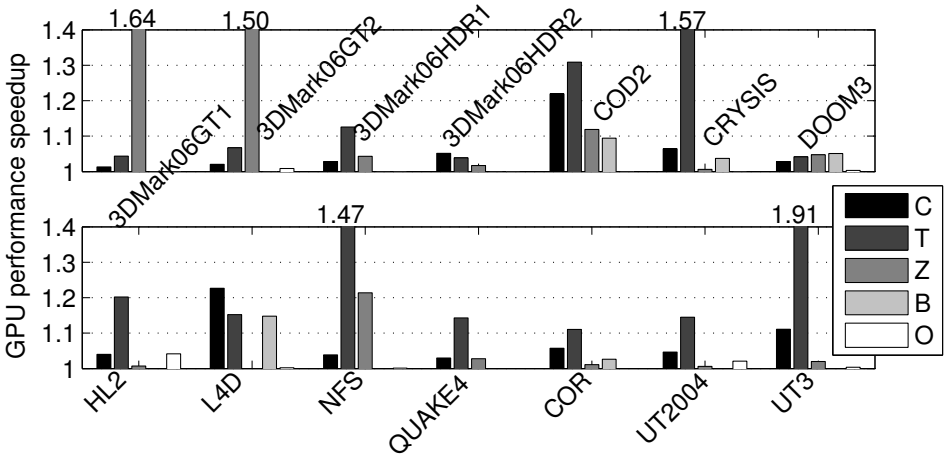


Figure 5.2: Speedup achieved when each individual stream is made to behave ideally.

We evaluate the performance-criticality of each stream by treating all non-compulsory LLC misses from that stream as hits. Figure 5.2 quantifies the speedup (ratio of frame rates with and without this optimization) achieved by accelerating each stream in this way. Performance-sensitivity of any particular stream varies

[1] Recall that the blitter is a special fixed function unit used to copy and process color data before it is sampled by the texture sampler.

across applications. A comparison of Figures 5.1 and 5.2 shows that the performance-sensitivity of the streams is not always in proportion to the volumes of DRAM accesses of the streams within an application. For example, in COD2, the depth accesses are more in number than the color accesses, but accelerating the color stream brings much higher speedup compared to accelerating the depth stream. In L4D, accelerating the color stream brings most benefits, but color accesses are much less in number compared to the texture accesses. In NFS, accelerating the texture accesses brings much bigger benefit than accelerating the depth accesses, although the access counts of these two streams are nearly equal.

Figure 5.3 quantifies the performance-criticality of a set of streams by treating all their non-compulsory LLC misses as hits. We focus on only a few sets for acceleration, namely, CT (set of color and texture), CTZ, CTZB, and CTZBO. The left bar ("COMBINED") for each application shows the stacked speedup as a new stream is added to the accelerated set starting from CT. For comparison, we also show the accumulated speedup when each stream in a set is individually accelerated in the bar "INDIVIDUAL". We observe that the combined speedup is much higher than the accumulated individual speedup in several applications. This indicates that there are certain inter-stream performance-dependencies that must be accelerated together. For example, a semantic dependence arises from the fact that the color and depth data may be consumed by the texture sampler for generating dynamic texture maps and shadow maps, respectively [26, 66]. On the other hand, accelerating the color stream without improving a bottlenecked texture stream may not be helpful because color blending is typically implemented after shading and texturing. We need to discover this inter-dependent critical group of streams at run-time.

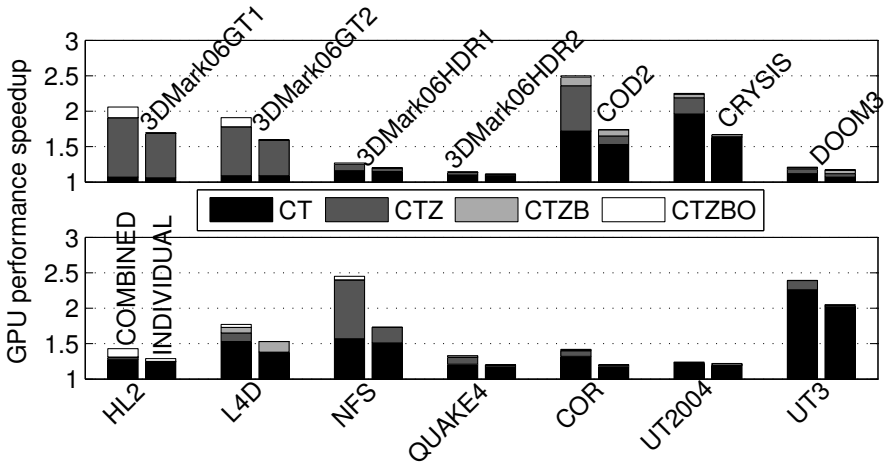


Figure 5.3: Speedup achieved when a set of streams is made to behave ideally.

5.1.2 Stream-centric Behavior in 3D Rendering Pipeline

To gain a better insight into the stream-centric criticality behavior shown by the 3D rendering applications, we conduct another experiment. In this experiment, on every ROP stall, we identify the stall sourcing unit (the unit where the next fragment to be processed is waiting) and record this event in a per stream counter where the stall sourcing unit is associated with the stream the unit generates. Figure 5.4 presents these results as a stacked bar chart showing percentage of stall cycles contributed by Z, T, and C streams, respectively. As the figure shows, stalls are distributed across streams. Since the expected performance improvement that can come from accelerating a stream is proportional to its stall contribution, a stream contributing a higher number of stall cycles can produce a higher gain than a stream contributing fewer stall cycles. Thus, criticality of a stream correlates directly with its stall contribution. We took this important insight and formalized it using a queuing network model discussed in Section 5.2.

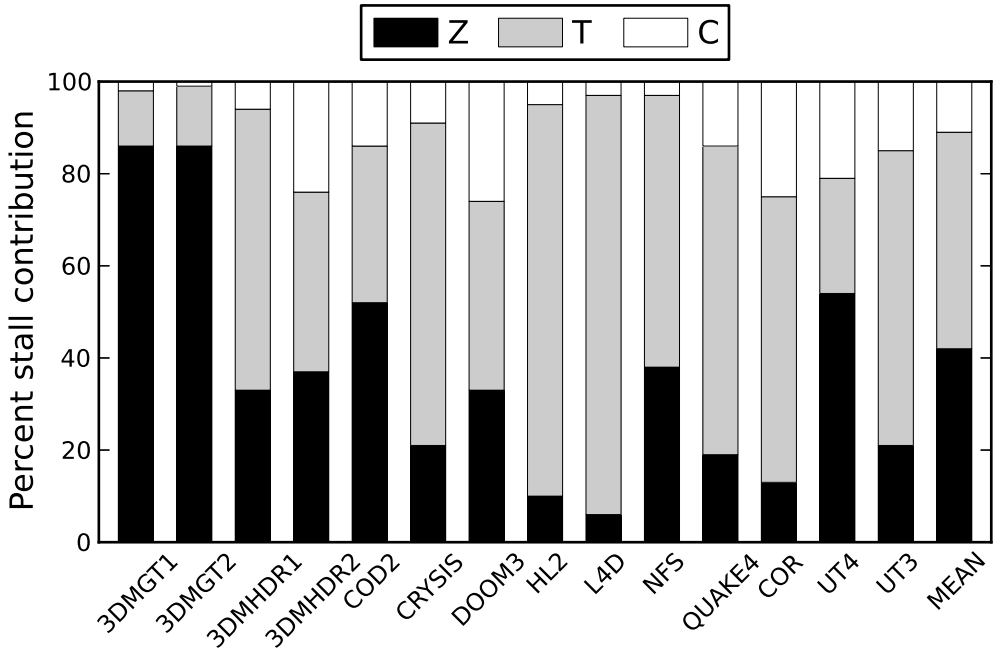


Figure 5.4: Stalls contributed by the Z, T, and C streams.

5.1.3 Stream Analysis of GPGPU Applications

For the GPGPU applications, the shader accesses constitute the dominant stream. We further divide the shader access stream based on the source static load/store instructions of the shader accesses. In this study, each static load/store shader instruction defines a distinct shader access stream. We adopt well-known stall-based techniques used in the CPU space for identifying the critical shader access streams [57, 67, 77]. More specifically, in each shader core, we maintain a fully-associative *stall table* with least-recently-used (LRU) replacement. Each entry of the table records the program counter (PC) of a shader instruction. If a shader instruction I stalls at dispatch time due to a pending operand, the parent shader instruction P that produces the operand is inserted into the stall table, provided P is a load instruction that has missed in the shader core's private cache. Subsequently, the accumulated stall cycle count introduced by P is tracked in its entry. The left

panel of Figure 5.5 shows the distribution of the DRAM accesses sourced by the shader instructions sorted by stall cycle count for six GPGPU workloads. "TnPC" denotes the top n shader instructions in this sorted list, while "All' denotes all load/store shader instructions. Top four shader instructions can cover almost all DRAM accesses except for LBM and CFD.

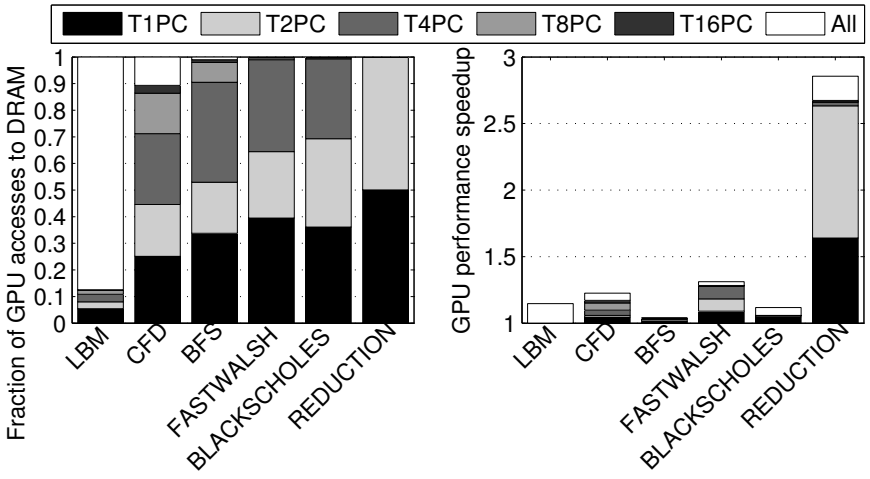


Figure 5.5: Left: distribution of DRAM accesses from GPGPU workloads. Right: speedup achieved when the top n PC streams are made to behave ideally.

The right panel of Figure 5.5 shows the speedup achieved when the load/store accesses sourced by the top n shader instructions are treated ideally in the LLC. We observe that the speedup data correlate well with the DRAM access distribution indicating that pipeline stall-based critical shader stream identification is a fruitful direction to pursue.

5.2 GPU Criticality-aware Memory Management

In this section, we present the GPU criticality-aware memory management proposal for heterogeneous CMPs. Section 5.2.1 discusses the mechanism for identifying the critical GPU access streams. The DRAM access scheduler presented in Section 5.2.2

shapes the priorities assigned to the CPU and GPU memory accesses.

5.2.1 Identifying Critical GPU Accesses

In the following, we present the mechanism used for identifying the critical accesses in 3D rendering and GPGPU workloads.

5.2.1.1 3D Scene Rendering Workloads

We represent the 3D rendering pipeline as an abstract queuing network of five units, namely, front-end (FE), depth/stencil test units (ZS), shader cores (SH), color blenders and writers (CW), and blitters (BT). The texture samplers are attached to the shader cores. The front-end loads vertex indices and vertex attributes, generates the geometry primitives, and produces the rasterized fragment quads.[1] The ZS unit removes the hidden surfaces based on a depth/stencil test on the fragments. The shader cores run a user-defined parallel shader program on each of the fragments received from the ZS unit. The shaded fragment quads are passed on to the CW unit for computing the final pixel color.[2] One ZS unit and one CW unit constitute one render output pipeline (ROP).

Queuing Model for Rendering Pipeline. We model the inter-dependence between the 3D rendering pipeline units using a queuing network shown in Figure 5.6. The model has $2n + 3$ queues, where n is the number of ROPs. The FE, SH, and BT units have one queue each. Each of the n ZS and CW units has one queue. Processing in the pipeline model can begin at FE or BT. In the first case, information

[1] A fragment quad is made of four fragments, each with complete information to render a pixel in the render buffer.

[2] In this discussion, we assume that depth/stencil test is done before pixel shading (known as early-Z). In certain situations, the ZS unit may have to be invoked after SH and before CW (known as late-Z).

flows through FE, ZS, SH, and CW in that order leading to the color output. This path gets activated during a traditional draw operation. The second path, which connects BT to the output, gets activated during the blitting process.

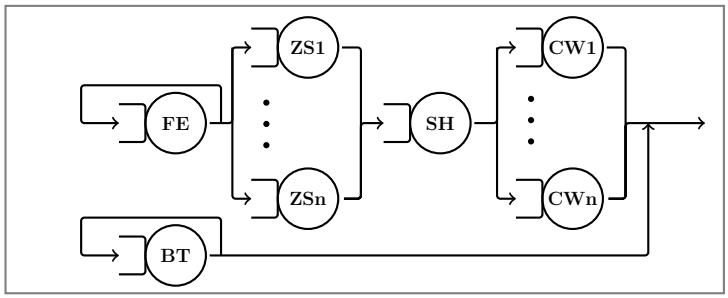


Figure 5.6: Queuing network for GPU pipeline.

Request Flow Monitoring. We monitor the request arrival and completion rates at each of the units to first identify the bottleneck units. For this purpose, we associate two up-down saturating counters $C_{in}[i]$ and $C_{out}[i]$ of width w bits to each unit i, where w is a configuration parameter (our evaluations use $w = 8$).[1] All counters are initialized to the mid-point i.e., 2^{w-1}. At the end of a cycle, $C_{in}[i]$ is incremented if unit i is found to have pending requests the count of which is above a threshold $th_{in}[i]$; otherwise $C_{in}[i]$ is decremented. Similarly, $C_{out}[i]$ is incremented, if unit i has completed more than a threshold, $th_{out}[i]$, number of requests; otherwise $C_{out}[i]$ is decremented. The peak input bandwidths of FE, ROP, and BT are used as $th_{in}[FE]$, $th_{in}[ROP]$, and $th_{in}[BT]$, respectively. Similarly, the peak output bandwidths determine $th_{out}[FE]$, $th_{out}[ROP]$, and $th_{out}[BT]$. We use the shader's peak input bandwidth divided by a constant[2] as $th_{in}[SH]$ as well as $th_{out}[SH]$.

Critical Stream Selection Algorithm. Using the values of $C_{in}[i]$ and $C_{out}[i]$, we first generate three bits for each unit i: $IOccupancy[i]$, $AOccupancy[i]$, and $Throughput[i]$. $IOccupancy[i]$ is 1 iff $C_{in}[i]$ of any instance of unit i is more than

[1] All algorithm parameters are tuned meticulously.

[2] The constant represents the number of cycles the shader program takes to process a fragment.

2^{w-1}. $AOccupancy[i]$ is 1 iff $C_{in}[i]$ for all instances of unit i are more than 2^{w-1}. $Throughput[i]$ is 1 iff $C_{out}[i]$ for all instances of unit i are more than 2^{w-1}. In general, if $Throughput[i]$ is 0 and $IOccupancy[i]$ is 1, the unit i is classified as bottlenecked and all accesses originating from it are classified as critical. For example, if SH is bottlenecked, all shader and texture sampler accesses would be marked critical. For SH to be bottlenecked, $Throughput[SH]$ must be 0 and $AOccupancy[SH]$ must be 1 meaning that throughput is low even though all shader units have enough work to do. If a unit i has multiple instances, e.g., ZS, SH, CW, and $AOccupancy[i]$ is 0, we identify the unit i as underloaded. To identify the bottleneck unit(s), we periodically execute Algorithm 1. First, this algorithm determines if CW and BT are bottlenecked. Next, it traverses the path FE-ZS-SH-CW (if early-Z is enabled) or the path FE-SH-ZS-CW (if early-z is disabled) from back to front. During this back-to-front traversal, if the algorithm encounters an underloaded unit U, it examines the unit V in front of U and finds out whether U is underloaded because V is bottlenecked.

5.2.1.2 GPGPU Workloads

For the GPGPU workloads, we employ a two-level algorithm invoked periodically for identifying the critical accesses. The first level of the algorithm identifies the bottlenecked shader cores. For each shader core, we maintain two saturating counters named $InputStall$ and $OutputStall$, each of width w bits ($w = 8$ in our implementation) and initialized to the mid-point i.e., 2^{w-1}. In a cycle, if the front-end of a shader core i fails to dispatch any warp due to pending source operands, the $InputStall[i]$ counter is incremented by one; otherwise it is decremented by one. Similarly, in a cycle, if the back-end of the shader core i fails to commit any shader instruction,

Algorithm 1 Algorithm to find bottleneck units

```
Inputs: IOccupancy (IO), AOccupancy (AO),
        Throughput (TH) vectors
Returns: Bottleneck vector
Initialize Bottleneck vector to zero.
if TH[CW] == 0 and IO[CW] == 1 then
    Bottleneck[CW] = 1
end if
if TH[BT] == 0 and IO[BT] == 1 then
    Bottleneck[BT] = 1
end if
                                        ▷ Back to front traversal
if CW is underloaded then
    if early-Z enabled then
        Check bottleneck unit using Algorithm 2
    else
        Check bottleneck unit using Algorithm 3
    end if
end if
```

the $OutputStall[i]$ counter is incremented by one; otherwise it is decremented by one. For a shader core i, if both $InputStall[i]$ and $OutputStall[i]$ are found to be above 2^{w-1}, the core is classified as bottlenecked. The second level of the algorithm employs the stall table introduced in Section 5.1 to identify the critical accesses from the bottlenecked cores. We use a sixteen-entry fully-associative LRU stall table per shader core. Among the instruction PC's captured by this table, the top few PC's covering up to 90% of the total stall cycles are considered to be generating critical accesses to the memory sub-system. If a load/store instruction misses in a bottlenecked shader core's private cache and is among the top few critical instructions captured by the stall table, the miss request sent to the LLC is marked critical. If such a shader instruction is not found in the stall table of a bottlenecked core, the request to the LLC is still marked critical, provided the LLC miss rate of GPU

Algorithm 2 Module to find bottleneck units in early-Z enable mode

```
if TH[SH] == 0 and AO[SH] == 1 then
    Bottleneck[SH] = 1
end if
if SH is underloaded then
    if TH[ZS] == 0 and IO[ZS] == 1 then
        Bottleneck[ZS] = 1
    end if
    if ZS is underloaded then
        Check FE state using Algorithm 4
    end if
end if
```

Algorithm 3 Module to find bottleneck units in early-Z disable mode

```
if TH[ZS] == 0 and IO[ZS] == 1 then
    Bottleneck[ZS] = 1
end if
if ZS is underloaded then
    if TH[SH] == 0 and AO[SH] == 1 then
        Bottleneck[SH] = 1
    end if
    if SH is underloaded then
        Check FE state using Algorithm 4
    end if
end if
```

accesses is at most 80%. In all other cases, the GPU access is marked non-critical. The non-critical shader accesses that miss in the LLC bypass the LLC freeing up space for other blocks.

5.2.2 Scheduling DRAM Accesses

The CPU and GPU requests that miss in the shared LLC access the DRAM. Every DRAM access coming from the GPU carries a bit specifying if the access is critical. We propose two DRAM scheduling policies, namely, the GPU-favoring policy and

Algorithm 4 Module to check FE bottleneck

```
if TH[FE] == 0 and IO[FE] == 1 then
    Bottleneck[FE] = Bottleneck[SH] = Bottleneck[ZS] = 1
end if
```

the interference mitigation policy (IM policy). The idea of prioritizing one kind of DRAM accesses over others has been proposed in the past [23, 37, 41, 100]. Our DRAM scheduling policy bears some similarity with these ideas. However, the manner in which we identify critical streams and mitigate interference due to critical stream prioritization adds novelty to our approach.

In the GPU-favoring policy, among the requests to the currently open row in a bank, the critical GPU accesses are served before considering the rest. When a new row needs to be activated in a bank, the oldest critical GPU access is given priority over the global oldest access. The GPU-favoring policy leads to two performance problems. First, the GPU fills arrive at a faster rate to the LLC replacing the CPU blocks earlier than the baseline. Second, the CPU requests may starve due to a long burst of critical GPU requests. The IM policy, designed to mitigate these problems, has two components, one to mitigate CPU starvation in the scheduler (IM-SCHED) and another to handle LLC interference (IM-LLC). While IM-SCHED is the default policy, a switch to IM-LLC takes place on detecting LLC interference.

The IM-SCHED component prioritizes CPU accesses over critical GPU accesses with a certain probability. The probability is obtained as follows. The execution is divided into equal intervals and at the end of each interval, the fraction of CPU requests de-prioritized by younger critical GPU requests during the interval is computed. This is used as the CPU prioritization probability for the next interval. If this probability is more than half, it is capped to half. This probability exceeds half

only in the GPGPU applications during 2-6% of all intervals.

For detecting LLC interference, the execution is divided into equal intervals and within an interval, the CPU applications are classified into high (H), medium (M), and low (L) intensities based on their LLC miss rates. The H category has more than 70% miss rate, the M category has miss rate between 10% and 70%, and the L category has miss rate at most 10%. In two consecutive intervals, if a CPU application's state is found to change from L to M or L to H which can be due to possible LLC interference, the application enters an emergency mode. The IM-LLC component is activated if there is at least one emergency mode CPU application. It schedules requests from emergency mode applications as often as critical GPU accesses. The remaining accesses are assigned lower priority. At the end of an interval, if an emergency mode application is found to go back to the L state, this indicates that the application benefits from IM-LLC. It continues to stay in the emergency mode. On the other hand, at the end of an interval, if an emergency mode application is still in M or H state, the application exits the emergency mode because it is not helpful for this application.

The CPU accesses are given higher priority than the non-critical GPU accesses except in one situation. In certain phases of the GPGPU workloads, the GPU becomes very sensitive to memory system performance. In these phases, it is possible to improve the GPU performance by sacrificing an equal amount of CPU performance and vice-versa. We decide to maintain the GPU performance in these phases by prioritizing all GPU accesses over the CPU accesses. To identify such phases, we periodically give the highest priority to all GPU accesses in the DRAM scheduler over a small time-window of 100K GPU cycles. If the GPGPU performance (shader instructions retired per cycle) improves during this window compared to the last

window, the scheduler continues to offer higher priority to all GPU accesses. The proposal on dynamic priority scheduler for heterogeneous mobile SoCs also gives highest priority to the GPU during the last 10% of a frame [37]. However, the purpose of our proposal offering highest priority to the GPU periodically is entirely different.

5.2.3 Additional Hardware Overhead

The critical stream identification logic needs to maintain the C_{in} and C_{out} counters for the FE, BT, ZS, SH, and CW units. The 3D rendering GPU models 64 SH units and sixteen ZS and CW units leading to 98 C_{in} and C_{out} counters requiring a total of 196 bytes. The GPGPU model has sixteen shader cores. Each core maintains one *OutputStall* counter, one *InputStall* counter, and a sixteen-entry stall table with each entry being 69 bits (32-bit PC, 32-bit stall cycles, one valid bit, and four LRU bits) amounting to 2.2 KB for all cores. The GPU criticality-aware policy is applied to the GPGPU applications and the 3D scene rendering applications that fail to meet a target FPS. As in the last chapter, we use 40 FPS as the target. To be able to identify the 3D scene rendering applications or application phases that fail to meet 40 FPS target, we keep our frame rate estimation hardware enabled. This hardware is borrowed from the previous chapter. The frame rate estimation mechanism maintains a 64-entry RTP information table, each entry being 97 bits. Overall, the storage overhead of our proposal is only 3.1 KB. Most importantly, none of the additional structures are accessed or updated on the critical path of execution. The structures that are accessed every cycle (such as the C_{in}, C_{out}, *InputStall*, and *OutputStall* counters) are small in size and expend energy much

smaller than what we save throughout the system (CMP die and DRAM device) by improving performance. The remaining structures are accessed less frequently and expend much lower energy.

5.3 Related Work

Memory access scheduling has been explored for CPU platforms, discrete GPU parts, and heterogeneous CMPs. The studies targeting the CPU platforms have attempted to improve the throughput as well as fairness of the threads that share the DRAM system [14, 18, 19, 23, 30, 34, 54, 56, 75, 76, 79, 87, 99, 100, 101]. Profile-guided assignment of DRAM channels to application groups has also been proposed [74]. Criticality estimation of load instructions [98] and criticality-driven memory access schedulers [23] for CPUs have been explored.

The memory access scheduling studies for the discrete GPU parts focus on the shader cores only and do not consider the rest of the GPU rendering pipeline. These studies have explored ways to minimize the latency variance among the threads within a warp [6], to accelerate the critical shader cores that do not have enough short-latency warps [41], and to design an appropriate mix of shortest-job-first and FR-FCFS with the goal of accelerating the less latency-tolerant shader cores [60]. There have been studies on warp and thread block schedulers for improving the memory system performance [2, 39, 40, 45, 61, 63].

Several studies have explored specialized memory access schedulers for heterogeneous systems [3, 37, 78, 95, 103]. The staged memory scheduler (SMS) clubs the memory requests from each source (CPU or GPU) into source-specific batches based on DRAM row locality [3]. Each batch is next scheduled with a probabilistic

mix of shortest-batch-first (favoring latency-sensitive jobs) and round-robin (enforcing fairness among bandwidth-sensitive jobs). The dynamic priority scheduler [37] proposed for mobile heterogeneous platforms employs dynamic progress estimation of tile-based deferred rendering (TBDR) [80, 86] and offers the GPU accesses equal priority as the CPU accesses if the GPU lags behind the target frame rendering time. Also, during the last 10% of the left time to render a frame, the GPU accesses are given higher priority than the CPU accesses. The subsequently proposed deadline-aware memory scheduler for heterogeneous systems (DASH) further improves the dynamic priority scheme by offering the highest priority to short-deadline applications and prioritizing the GPU applications that lag behind the target [103]. The option of statically partitioning the physical address space between the CPU and GPU datasets and assigning two independent memory controllers to handle accesses to the two datasets has been explored [78]. A subsequent study has shown that such static partitioning of memory resources can be sub-optimal [46].

Insertion and replacement policies to manage the shared LLC in the heterogeneous CMPs have been explored [62, 85]. Selective LLC bypass policies for GPU misses arising from latency-tolerant shader cores have been proposed [71].

5.4 Simulation Results

We evaluate our proposal on a simulated heterogeneous CMP with four CPU cores and one GPU. We use a bigger set of heterogeneous workload mixes in this study compared to the previous two studies. The set of fourteen 3D scene rendering jobs is same as in the previous studies. We use six GPGPU applications shown in Table 5.1. We select thirteen SPEC CPU 2006 applications and partition them into two groups

Table 5.1: CUDA application details

Application	Thread configuration
LBM	120×150 blocks, 120 threads/block
CFD	759 blocks, 128 threads/block
BFS	1954 blocks, 512 threads/block
FASTWALSH	8192 blocks, 256 threads/block
BLACKSCHOLES	480 blocks, 128 threads/block
REDUCTION	64 blocks, 256 threads/block

based on the LLC misses per kilo instructions (MPKI). The high MPKI group (H-group) contains bwaves, lbm, leslie3d, libquantum, mcf, milc, and soplex. The low MPKI group (L-group) contains bzip2, gcc, omnetpp, sphinx3, wrf, and zeusmp. Each of the twenty GPU workloads (fourteen 3D rendering and six GPGPU) is co-executed with three different four-way multi-programmed CPU workload mixes. To do this, we use the applications from the H-group to prepare twenty four-way H mixes. Similarly, we prepare twenty four-way L mixes from the L-group. We also prepare twenty four-way HL mixes, each of which has two H-group and two L-group applications. Each of the twenty GPU workloads is mixed with one CPU mix each from the H, L, and HL sets. We evaluate our proposal on these sixty different heterogeneous mixes executed on a CMP with four CPU cores and a GPU. For each GPU workload, we report the performance averaged (geometric mean) over the three mixes containing that GPU workload. Sections 5.4.1 and 5.4.2 respectively discuss the results for the mixes containing the 3D rendering and CUDA workloads.

5.4.1 Mixes with 3D Rendering Workloads

We divide the discussion into evaluation of the several individual components that constitute our proposal.

Critical vs. Non-critical Accesses. We conduct two experiments to understand whether our critical access identification logic is able to mark the critical GPU accesses as such. In one case, we treat all non-compulsory LLC misses from the critical accesses as hits. In the other case, we treat all non-compulsory LLC misses from the non-critical accesses as hits. Figure 5.7 shows the improvement in FPS over the baseline in the two cases. Except for L4D, all applications show much higher FPS improvement when the critical accesses are treated ideally. These results confirm that our proposal is able to identify a subset of the critical accesses correctly. On average, treating the critical accesses ideally offers an FPS improvement of 48%, while favoring the complementary access set offers only 13% improvement. In L4D, our algorithm misclassifies a number of critical blitter accesses. COR loses performance when the non-critical accesses are treated ideally because some of the non-critical accesses negatively interfere with the critical ones.

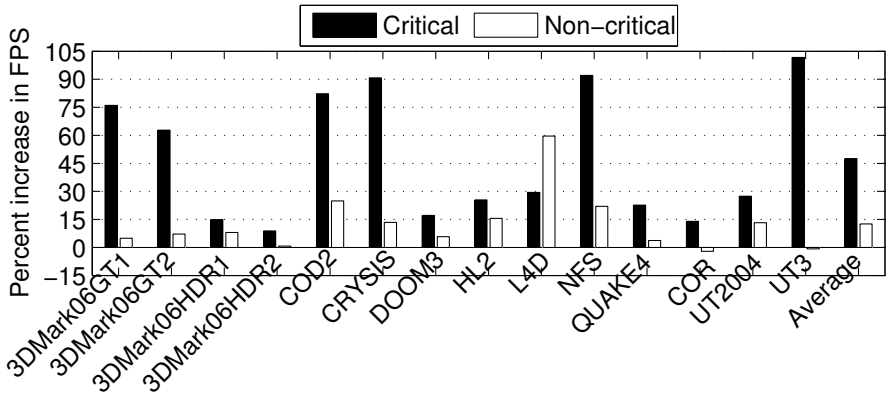


Figure 5.7: Percent improvement in FPS when LLC behaves ideally for critical and non-critical accesses.

Figure 5.8 shows the distribution of the critical color (C), critical texture (T), critical depth (Z), critical blitter (B), critical other (O), and non-critical (NC) accesses as identified by our algorithm in the aforementioned experiment. The distribution varies widely across the applications with 62% of accesses being identified as critical on average. It is encouraging to note that for most of the applications, the stream that was found to enjoy the largest speedup in Figure 5.2 is among the dominant critical streams identified by our algorithm.

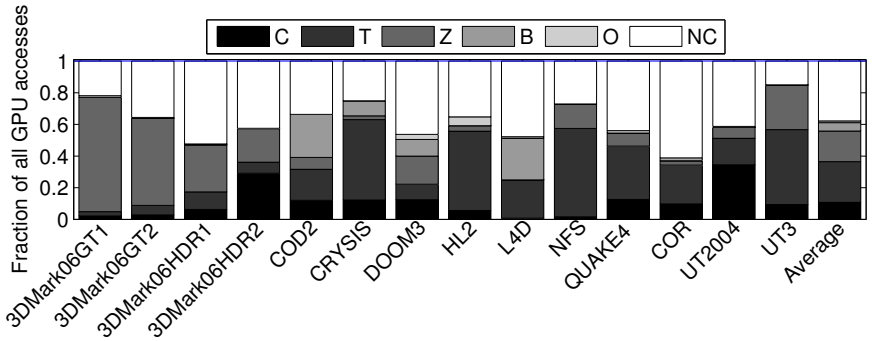


Figure 5.8: Distribution of critical accesses.

DRAM Scheduling for Critical GPU Accesses. Our DRAM scheduling proposal employs the access criticality information for the 3D rendering applications that fail to meet a target FPS. We set this target to 40 FPS and show the results for the eight applications that deliver frame rate below this level (see the last column of Table 2.2). Figure 5.9 evaluates the GPU-favoring and IM policies (Section 5.2.2) for the mixes containing these GPU applications. The left panel shows the FPS of the GPU normalized to the baseline. The right panel shows the weighted speedup for the corresponding CPU mixes normalized to the baseline. We identify each CPU workload by *GPUworkloadnameCPU*. The GPU-favoring policy improves the FPS by 18% on average while degrading the weighted speedup of the CPU mixes by 8% on average. The IM policy is able to recover most of the lost CPU performance.

This policy improves the FPS of the GPU applications by 15% on average while performing within 3% of the baseline for the CPU application mixes. The CPU mixes co-scheduled with 3DMark06HDR1 perform better than the baseline, on average. The IM policy has the IM-SCHED and IM-LLC components. Compared to the GPU-favoring policy, the IM-LLC component alone reduces CPU performance loss by 3% while sacrificing 2% GPU performance. The IM-SCHED component alone reduces CPU performance loss by 2% while sacrificing 1% GPU performance. Effects are additive when they work together in the IM policy.

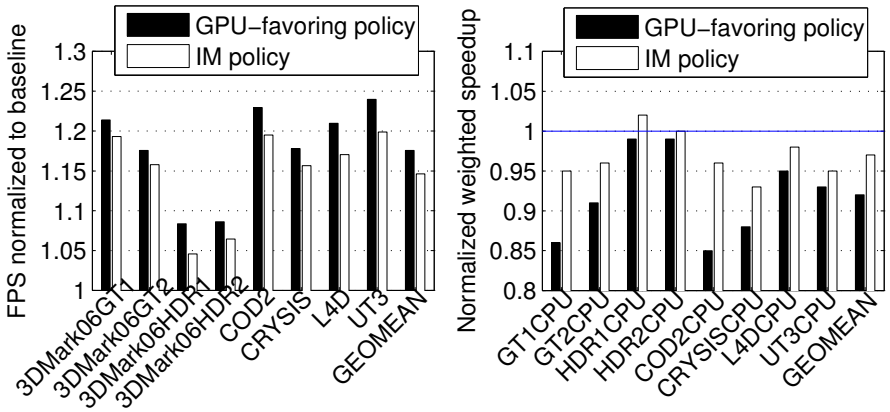


Figure 5.9: Left: normalized FPS of GPU applications that perform below target FPS. Right: weighted CPU speedup for the mixes.

Comparison to Related Proposals. We compare our proposal against staged memory scheduling (SMS) [3], dynamic priority scheduler (DynPrio) [37], and deadline-aware scheduling (DASH) [103]. These proposals were discussed in Section 5.3. We evaluate two versions of SMS, namely, one with a probability of 0.9 of using shortest-job-first (SMS-0.9) and the other with this probability zero (SMS-0) i.e., it always selects the round-robin policy. DynPrio and DASH make use of our frame rate estimation technique to compute the time left in a frame. Additionally, we compare our proposal against HeLM, the state-of-the-art shared LLC management policy for

heterogeneous CMPs [71].

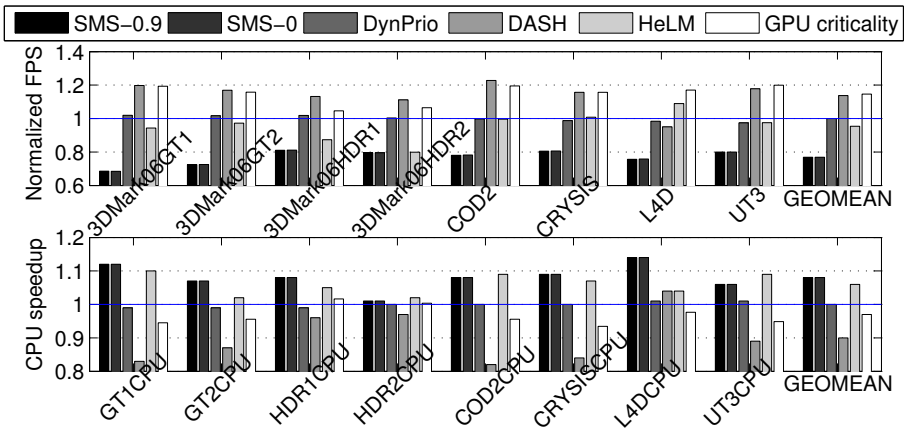


Figure 5.10: Top: FPS speedup over baseline. Bottom: weighted CPU speedup for the mixes.

Figure 5.10 shows the comparison for the heterogeneous mixes containing the GPU applications that fail to meet the target FPS. SMS suffers from large losses in FPS (upper panel) due to the delay in batch formation. DynPrio fails to observe any overall benefit because it offers express bandwidth to the GPU application only during the last 10% of a frame time. Both DASH and our GPU criticality-aware proposal (IM policy) improve average FPS by 14%. DASH prioritizes the GPU accesses throughout the execution. Such a policy, however, hurts the performance of the co-scheduled CPU mixes by 10% on average (lower panel of Figure 5.10). Our proposal, on the other hand, accelerates only the critical GPU accesses and improves average FPS by the same amount as DASH while delivering CPU performance within 3% of the baseline. Both SMS-0.9 and SMS-0 improve CPU mix performance by 8%, while suffering from large losses in GPU performance. HeLM improves CPU performance by 6% on average, while degrading GPU performance by 5%. To understand how these proposals fare in terms of combined CPU-GPU system performance, we consider a performance metric in which the CPU and the GPU performance are weighed equally i.e., overall speedup is the geometric mean of the FPS speedup and the nor-

malized weighted speedup of the CPU mix [62]. We find that DASH and HeLM improve this performance metric by 1% on average compared to the baseline, while our proposal improves this metric by 5%. DynPrio delivers baseline performance, while both SMS-0.9 and SMS-0 degrade the equal-weight metric by 9%.

Sensitivity to LLC Capacity. Figure 5.11 summarizes the performance of the IM policy when the heterogeneous CMP is equipped with an 8 MB shared LLC (as opposed to 16 MB considered so far). The GPU applications improve by an impressive 17% over the baseline and the co-scheduled CPU application mixes perform within 4% of the baseline, on average. The CPU mixes co-scheduled with 3DMark06HDR1 and 3DMark06HDR2 outperform the baseline, on average. Referring back to Figure 5.9, we observe that for a 16 MB LLC, the GPU gain is 15% and the CPU mixes perform within 3% of the baseline, on average.

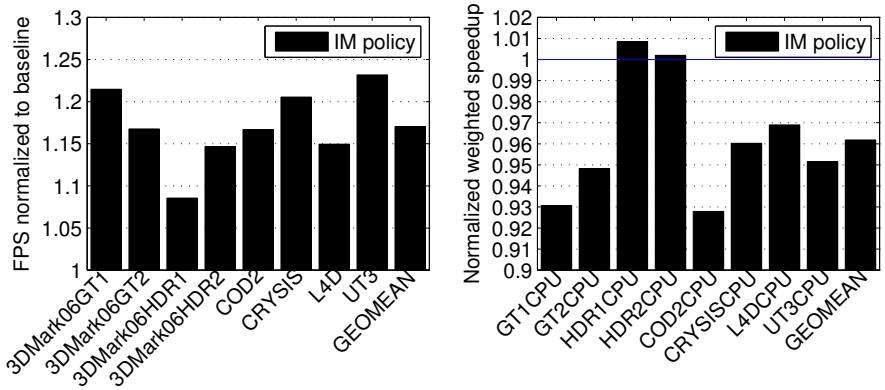


Figure 5.11: Left: normalized FPS of GPU applications that perform below target FPS. Right: weighted CPU speedup for the mixes.

5.4.2 Mixes with GPGPU Workloads

Figure 5.12 evaluates SMS-0.9, SMS-0, HeLM, and our GPU criticality-aware proposal for the heterogeneous mixes containing CUDA applications when the CMP is

equipped with a 16 MB shared LLC.[1] Both SMS-0.9 and SMS-0 degrade GPU performance (left panel) by 4% on average while improving the CPU performance (right panel) by 7% and 8%, respectively. HeLM improves GPU performance by 6% and CPU performance by 7%, on average. Our proposal improves GPU performance by 1% and CPU performance by 14%, on average. Since the GPU performance can be traded off for CPU performance and vice-versa, we use the equal-weight performance metric to understand the overall system performance. Both SMS-0.9 and SMS-0 improve the equal-weight metric by 2%, while HeLM improves this metric by 6%. Our proposal achieves a 7% improvement in this metric.

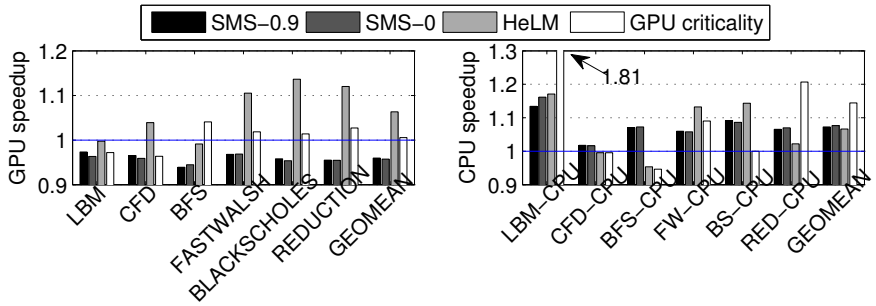


Figure 5.12: Left: GPU application speedup. Right: weighted CPU speedup for the mixes.

5.5 Conclusions

We have presented a new class of memory access schedulers for heterogeneous CMPs. Our proposal dynamically identifies the critical GPU accesses and probabilistically prioritizes them in the memory access scheduler. Detailed simulation studies show that our proposal achieves its goal of offering a bigger share of the shared memory system resources to the critical GPU accesses. The GPU performance improves

[1] DynPrio and DASH are left out from this evaluation because these two proposals are suitable for deadline-sensitive GPU workloads.

by 15% on average for the 3D scene rendering applications, while the co-scheduled CPU application mixes perform within 3% of the baseline on average. For the heterogeneous mixes with GPGPU applications, the CPU application mixes improve by 14% on average, while the GPU performs 1% above the baseline leading to an overall 7% improvement in system performance, measured in terms of a CPU-GPU equal-weight performance metric.

Chapter 6

Summary and Future Work

In this dissertation, we have presented solutions to improve the memory system performance of a single-chip CPU-GPU heterogeneous processor design. Our target architecture is similar to the Intel's integrated GPU architectures, sharing resources, such as, last-level cache, on-chip interconnect, on-chip memory controller, and DRAM channels, ranks, banks between the two type of cores. We have explored heterogeneous workload scenarios where general purpose CPU applications (drawn from the SPEC CPU 2006 suite) and 3D scene rendering or GPGPU applications are simultaneously executed on the CPU and GPU cores, respectively. Our experiments show that, in such an environment, both CPU and GPU applications degrade significantly from their standalone performance due to the contention for the shared memory system resources. The case study with Intel Haswell presented in Chapter 1 confirms that such problems exist in modern commercial heterogeneous processors.

Since the LLC capacity and DRAM bandwidth are the two major resources shared between the CPU and GPU cores in a heterogeneous environment like the one studied in this dissertation, we explore LLC capacity, DRAM bandwidth, and

their collective management policies covering various heterogeneous execution use cases. We find that efficient LLC capacity management not only improves memory access latency but also helps alleviate bandwidth pressure in the shared DRAM system. Performance feedback-directed dynamic partitioning of DRAM bandwidth between CPU and GPU applications is another solution space that we explore to deal with the DRAM interference problem. Further, we show that in certain scenarios, the memory access rate of the GPU can be controlled to improve memory system resource allocation in a heterogeneous processor.

Our LLC management policy that dynamically partitions cache capacity between CPU and GPU applications is discussed in Chapter 3. Since these applications differ vastly in reuse characteristics and access patterns, for efficient management, these applications need to be handled appropriately. Moreover, we find that the GPU applications exhibit large working sets, and as a result, effectively learning their reuse patterns is difficult. To this end, we propose a novel working set sample cache and employ it to effectively learn dynamic reuse probability of CPU and GPU access streams. The measured reuse probability is further used to induce an implicit partition of the LLC capacity between the CPU and GPU access streams. Evaluation done on a heterogeneous processor simulator shows that with a 16 MB shared last-level cache we are able to improve the performance of the GPU workloads spanning DirectX and OpenGL as well as CUDA applications by 12% on average and of the co-scheduled quad-core CPU workloads by 7% on average.

3D scene rendering applications executing on GPU have well-defined QoS requirements (i.e., target frame rate) for visual satisfaction of the end-users. If such an application is able to meet the required level of performance, memory resources can be shifted from GPU to CPU by controlling the memory access rate of the

GPU. In Chapter 4, we present our QoS-guided dynamic GPU access throttling algorithm. Our proposal uses an accurate frame rate prediction algorithm that dynamically measures the QoS of the GPU applications and a light-weight memory access throttling mechanism that dynamically controls the memory access rate of the GPU application. Our proposal is able to effectively transfer the LLC capacity and DRAM bandwidth from GPU to CPU cores whenever the GPU application is able to meet the target QoS level. Evaluation done on a simulated heterogeneous configuration having four CPU cores and a GPU shows that we are able to improve the CPU performance by 18% on average while effectively meeting the QoS target for the GPU applications.

Our GPU criticality-driven memory management proposal presented in Chapter 5 deals with the problem of dynamic partitioning of the DRAM bandwidth between CPU and GPU access streams. We find that the accesses originating from different parts of the rendering pipeline or different shader load/store instructions are not equally important for end-performance. To dynamically learn this criticality information, we propose a queuing network to model the information flow through the rendering pipeline. This model forms the central contribution of our proposal. For estimating the criticality of shader load/store instructions we make use of the front-end and back-end shader core pipeline stalls to design a two-level algorithm. We use the measured criticality information to partition the DRAM bandwidth between the critical GPU, non-critical GPU, and CPU accesses. Evaluation done on a heterogeneous chip-multiprocessor simulator having four CPU cores and a GPU shows that for the 3D scene rendering applications, our proposal is able to improve the GPU performance by 15% on average without degrading the performance of the co-scheduled CPU applications by much. For the GPGPU applications, our

proposal improves the system performance by 7% on average.

6.1 Future Work

The heterogeneous processors available today enable tight coupling between the CPU and GPU cores. Introduction of shared virtual memory, system wide atomics, and global coherence in such processors allow partitioning of data and instruction between CPU and GPU cores at various granularity leading to fine-grain data sharing between the CPU and the GPU. Our work can be extended to deal with the memory system interference in such fine-grain partitioned heterogeneous applications for improving the overall system performance.

The studies presented in this dissertation show that the working set of the GPU applications is very large, far exceeding the capacity of the modern on-chip SRAM caches. Systems employing large high-bandwidth die-stacked or embedded last-level DRAM caches are attractive for workloads having large working sets with high bandwidth demands. It is known that effectively managing the bandwidth and capacity of the DRAM cache in such systems is an important problem. Our proposals can be effectively extended to tackle such problems.

Different kinds of cores (CPU and GPU) feature very different execution models and thus stress the memory system in very different ways. A CPU needs to perform memory reads with low latency, while a GPU can tolerate long memory latency due to the massively parallel design with thousands of ready contexts, but requires significantly large memory system bandwidth. A hybrid memory system combining high-bandwidth non-volatile memory (NVM) with traditional DDRx memory and a low-latency SRAM cache can be used to improve the overall system performance

and power in CPU-GPU heterogeneous processors. Our proposals can be suitably augmented to work in such hybrid memory systems.

References

[1] K. Asanovic et al. A View of the Parallel Computing Landscape. In *Communications of the ACM*, **52**(10): 56–67, October 2009.

[2] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38, October 2015.

[3] R. Ausavarungnirun, K. K-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 416–427, June 2012.

[4] L. A. Belady. A Study of Replacement Algorithms for a Virtual-storage Computer. In *IBM Systems Journal*, **5**(2): 78–101, 1966.

[5] D. Bouvier, B. Cohen, W. Fry, S. Godey, and M. Mantor. Kabini: An AMD Accelerated Processing Unit System on a Chip. In *IEEE Micro*, **34**(2):22–33, March/April 2014.

[6] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian. Managing DRAM Latency Divergence in Irregular GPGPU Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 128–139, November 2014.

[7] M. Chaudhuri et al. Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches. In *Proceedings of the 21st International Conference on Parallel Architecture and Compilation Techniques*, pages 293–304, September 2012.

[8] M. Chaudhuri. Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches. In *Proceedings of the 42nd International Symposium on Microarchitecture*, pages 401–412, December 2009.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 44–54, October 2009.

[10] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 1–11, December 2010.

[11] X. Chen et al. Adaptive Cache Management for Energy-efficient GPU Computing. In *Proceedings of the 47th International Symposium on Microarchitecture*, pages 343–355, December 2014.

[12] C. J. Choi et al. Performance Comparison of Various Cache Systems for Texture Mapping. In *Proceedings of the 4th International Conference on High Performance Computing in Asia-Pacific Region*, pages 374–379, May 2000.

[13] M. Cox, N. Bhandari, and M. Shantz. Multi-level Texture Caching for 3D Graphics Hardware. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 86–97, June/July 1998.

[14] 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 *Proceedings of the 19th International Symposium on High Performance Computer Architecture*, pages 107–118, February 2013.

[15] M. Demler. Iris Pro Takes On Discrete GPUs. In *Microprocessor Report*, September 9, 2013.

[16] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Optimization Framework for Bulk-synchronous Applications in Heterogeneous Systems. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques*, pages 353–364, September 2010.

[17] N. Doung et al. Improving Cache Management Policies Using Dynamic Reuse Distances. In *Proceedings of the 45th International Symposium on Microarchitecture*, pages 389–400, December 2012.

[18] 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 *Proceedings of the 15th International Conference on Ar-*

chitectural Support for Programming Languages and Operating Systems, pages 335–346, March 2010.

[19] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt. Parallel Application Memory Scheduling. In *Proceedings of the 44thInternational Symposium on Microarchitecture*, pages 362–373, December 2011.

[20] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th International Symposium on Computer Architecture*, pages 365–376, June 2011.

[21] J. Gaur et al. Efficient Management of Last-level Caches in Graphics Processors for 3D Scene Rendering Workloads. In *Proceedings of the 46th International Symposium on Microarchitecture*, pages 395–407, December 2013.

[22] J. Gaur, M. Chaudhuri, and S. Subramoney. Bypass and Insertion Algorithms for Exclusive Last-level Caches. In *Proceedings of the 38th International Symposium on Computer Architecture*, pages 81–92, June 2011.

[23] S. Ghose, H. Lee, and J. F. Martinez. Improving Memory Scheduling via Processor-side Load Criticality Information. In *Proceedings of the 40th International Symposium on Computer Architecture*, pages 84–95, June 2013.

[24] N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer Visibility. In *Proceedings of the 20th SIGGRAPH Annual Conference on Computer Graphics and Interactive Techniques*, pages 231–238, August 1993.

[25] Z. S. Hakura and A. Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 108–120, May 1997.

[26] M. Harris. Dynamic Texturing. Available at http://developer.download.nvidia.com/assets/gamedev/docs/ DynamicTexturing.pdf.

[27] HP Labs. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. Available at http://www.hpl.hp.com/research/mcpat/.

[28] Joshua Ho, Ryan Smith. NVIDIA Tegra Xl Preview and Architecture Analysis. Available at https://www.anandtech.com/show/8811/nvidia-tegra-x1-preview.

[29] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 209–220, May 2002.

[30] I. Hur and C. Lin. Adaptive History-Based Memory Schedulers. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 343–354, December 2004.

[31] H. Igehy, M. Eldridge, and P. Hanrahan. Parallel Texture Caching. In *Proceedings of the SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 95–106, August 1999.

[32] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a Texture Cache Architecture. In *Proceedings of the SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 133–142, August/September 1998.

[33] Intel Corporation. Intel Core i7-4770 Processor. Available at http://ark.intel.com/products/75122/Intel-Core-i7-4770-Processor-8M-Cache-up-to-3_90-GHz.

[34] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 39–50, June 2008.

[35] A. Jaleel et al. High Performance Cache Replacement using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th International Symposium on Computer Architecture*, pages 60–71, June 2010.

[36] A. Jaleel et al. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques*, pages 208–219, October 2008.

[37] M. K. Jeong, M. Erez, C. Sudanthi, and N. C. Paver. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MP-SoC. In *Proceedings of the 49th Annual Design Automation Conference*, pages 850–855, June 2012.

[38] W. Jia, K. A. Shaw, and M. Martonosi. MRPB: Memory Request Prioritization for Massively Parallel Processors. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, pages 272–283, February 2014.

[39] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the 40th International Symposium on Computer Architecture*, pages 332–343, June 2013.

[40] A. Jog, O. Kayiran, 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 *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 395–406, March 2013.

[41] 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 *Proceedings of the International Conference on Measurement and Modeling of Computer Science (SIGMETRICS)*, pages 351–363, June 2016.

[42] D. Kanter. Intel's Ivy Bridge Graphics Architecture. April 2012. Available at http://www.realworldtech.com/ivy-bridge-gpu/.

[43] D. Kanter. Intel's Sandy Bridge Graphics Architecture. August 2011. Available at http://www.realworldtech.com/sandy-bridge-gpu/.

[44] D. Kanter. AMD Fusion Architecture and Llano. June 2011. Available at http://www.realworldtech.com/fusion-llano/.

[45] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 157–166, September 2013.

[46] O. Kayiran, 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 *Proceedings of the 47th International Symposium on Microarchitecture*, pages 114–126, December 2014.

[47] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache Replacement Based on Reuse Distance Prediction. In *Proceedings of the 25th International Conference on Computer Design*, pages 245–250, October 2007.

[48] S. Khan, A. Alameldeen, C. Wilkerson, O. Mutlu, and D. A. Jimènez. Improving Cache Performance by Exploiting Read-Write Disparity. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, pages 452–463, February 2014.

[49] S. Khan and D. A. Jimènez. Insertion Policy Selection Using Decision Tree Analysis. In *Proceedings of the 28th International Conference of Computer Design*, pages 106–111, October 2010.

[50] S. Khan, Y. Tian, and D. A. Jimènez. Dead Block Replacement and Bypass with a Sampling Predictor. In *Proceedings of the 43rd International Symposium on Microarchitecture*, pages 175–186, December 2010.

[51] S. Khan, Z. Wang, and D. A. Jimènez. Decoupled Dynamic Cache Segmentation. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, pages 235–246, February 2012.

[52] S. Khan et al. Using Dead Blocks as a Virtual Victim Cache. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 489–500, September 2010.

[53] M. Kharbutli and Y. Solihin. Counter-based Cache Replacement and Bypassing Algorithms. In *IEEE Transactions on Computers*, **57**(4): 433–447, April 2008.

[54] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers. In *Proceedings of the 16th International Conference on High-Performance Computer Architecture*, January 2010.

[55] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho. MacSim: A CPU-GPU Heterogeneous Simulation Framework. February 2012. Available at https://code.google.com/p/macsim/.

[56] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *Proceedings of the 43rd International Symposium on Microarchitecture*, pages 65–76, December 2010.

[57] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez. Checkpointed Early Load Retirement. In *Proceedings of the 11th International Conference on High-Performance Computer Architecture*, pages 16–27, February 2005.

[58] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. 2004. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *In Proceedings of the 31st annual international symposium on Computer architecture* (ISCA '04). IEEE Computer Society, Washington, DC, USA, 64-.

[59] A-C. Lai, C. Fide, and B. Falsafi. Dead-block Prediction & Dead-block Correlating Prefetchers. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 144–154, June/July 2001.

[60] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin. DRAM Scheduling Policy for GPGPU Architectures Based on a Potential Function. In *IEEE Computer Architecture Letters*, **11**(2): 33–36, July 2012.

[61] S-Y. Lee, A. Arunkumar, and C-J. Wu. CAWA: Coordinated Warp Scheduling and Cache Prioritization for Critical Warp Acceleration of GPGPU Workloads. In *Proceedings of the 42nd International Symposium on Computer Architecture*, pages 515–527, June 2015.

[62] J. Lee and H. Kim. TAP: A TLP-aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, pages 91–102, February 2012.

[63] S-Y. Lee and C-J. Wu. CAWS: Criticality-aware Warp Scheduling for GPGPU Workloads. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 175–186, August 2014.

[64] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder. Priority-based Cache Allocation in Throughput Processors. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture*, pages 89–100, February 2015.

[65] H. Liu et al. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *Proceedings of the 41st International Symposium on Microarchitecture*, pages 222–233, November 2008.

[66] F. D. Luna. *Introduction to 3D Game Programming with DirectX 10*. Wordware Publishing Inc..

[67] R. Manikantan and R. Govindarajan. Focused Prefetching: Performance Oriented Prefetching Based on Commit Stalls. In *Proceedings of the 22nd International Conference on Supercomputing*, pages 339–348, June 2008.

[68] R. Manikantan, K. Rajan, and R. Govindarajan. Probabilistic Shared Cache Management (PriSM). In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 428–439, June 2012.

[69] R. Manikantan, K. Rajan, and R. Govindarajan. NUcache: An Efficient Multicore Cache Organization Based on Next-Use Distance. In *Proceedings of the 17th IEEE International Symposium on High-performance Computer Architecture*, pages 243–253, February 2011.

[70] R. L. Mattson et al. Evaluation Techniques for Storage Hierarchies. In *IBM Systems Journal*, **9**(2): 78–117, 1970.

[71] V. Mekkat et al. Managing Shared Last-level Cache in a Heterogeneous Multicore Processor. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 225–234, September 2013.

[72] P. Messmer. Interactive Supercomputing with In-Situ Visualization on Tesla GPUs. Available at https://devblogs.nvidia.com/parallelforall/interactive-supercomputing-in-situ-visualization-tesla-gpus/.

[73] V. Moya et al. ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 231–241, March 2006. Source and traces available at http://attila.ac.upc.edu/wiki/index.php/Main_Page.

[74] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. T. Kandemir, and T. Moscibroda. Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning. In *Proceedings of the 44th International Symposium on Microarchitecture*, pages 374–385, December 2011.

[75] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of the 40th International Symposium on Microarchitecture*, pages 146–160, December 2007.

[76] O. Mutlu and T. Moscibroda. Parallelism-aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 63–74, June 2008.

[77] 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 *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 129–140, February 2003.

[78] N. C. Nachiappan, P. Yedlapalli, N. Soundararajan, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das. GemDroid: A Framework to Evaluate Mobile Platforms. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 355–366, June 2014.

[79] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. 2006. Fair Queuing Memory Systems. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 208–222, December 2006.

[80] T. Olson. Mali 400 MP: A Scalable GPU for Mobile and Embedded Devices. In *Symposium on High-Performance Graphics*, June 2010.

[81] David A. Patterson and John L. Hennessy. 1990. Computer Architecture: a Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[82] T. Piazza. Intel Processor Graphics. In *Symposium on High-Performance Graphics*, August 2012.

[83] M. K. Qureshi et al. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 381–391, June 2007.

[84] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 423–432, December 2006.

[85] S. Rai and M. Chaudhuri. Exploiting Dynamic Reuse Probability to Manage Shared Last-level Caches in CPU-GPU Heterogeneous Processors. In *Proceedings of the 30th International Conference on Supercomputing*, June 2016.

[86] M. Ribble. Next-gen Tile-based GPUs. In *Game Developers' Conference*, 2008.

[87] S. Rixner, W. J. Dally, U. J. Kapasi, P. R. Mattson, and J. D. Owens. Memory Access Scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 128–138, June 2000.

[88] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware Warp Scheduling. In *Proceedings of the 46th International Symposium on Microarchitecture*, pages 99–110, December 2013.

[89] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. In *IEEE Computer Architecture Letters*, **10**(1): 16–19, January-June 2011.

[90] D. Sanchez and C. Kozyrakis. Vantage: Scalable and Efficient Fine-grain Cache Partitioning. In *Proceedings of the 38th International Symposium on Computer Architecture*, pages 57–68, June 2011.

[91] 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 *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 355–366, September 2012.

[92] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International*

Conference on Architectural Support for Programming Languages and Operating Systems, pages 45–57, October 2002.

[93] A. L. Shimpi. Intel Iris Pro 5200 Graphics Review: Core i7-4950HQ Tested. June 2013. Available at http://www.anandtech.com/show/6993/intel-iris-pro-5200-graphics-review-core-i74950hq-tested.

[94] D. Shingari, A. Arunkumar, and C-J. Wu. Characterization and Throttling-Based Mitigation of Memory Interference for Heterogeneous Smartphones. In *Proceedings of the International Symposium on Workload Characterization*, pages 22–33, October 2015.

[95] A. Stevens. QoS for High-performance and Power-efficient HD Multimedia. *ARM White Paper*, 2010.

[96] J. Stone. HPC Visualization on Nvidia Tesla GPUs. Available at https://devblogs.nvidia.com/parallelforall/hpc-visualization-nvidia-tesla-gpus/.

[97] J. A. Stratton, C. Rodrigues, I-J. Sung, N. Obeid, L-W. Chang, N. Anssari, G. D. Liu, and W-m. W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *IMPACT Technical Report IMPACT-12-01*, March 2012.

[98] S. Subramaniam, A. Bracy, H. Wang, and G. H. Loh. Criticality-based Optimizations for Efficient Load Processing. In *Proceedings of the 15th International Conference on High-Performance Computer Architecture*, pages 419–430, February 2009.

[99] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu. The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost. In *Proceedings of the 32nd International Conference on Computer Design*, pages 8–15, October 2014.

[100] L. Subramanian, V. Seshadri, A. Ghosh, S. M. Khan, and O. Mutlu. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 62–75, December 2015.

[101] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In *Proceedings of the 19th International Symposium on High Performance Computer Architecture*, pages 639–650, February 2013.

[102] R. Ubal et al. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the 21st International Conference on Parallel Architecture and Compilation Techniques*, pages 335–344, September 2012.

[103] H. Usui, L. Subramanian, K. K-W. Chang, and O. Mutlu. DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. In *ACM Transactions on Architecture and Code Optimization*, **12**(4), January 2016.

[104] A. Vartanian, J-L. Bechennec, and N. Drach-Temam. Evaluation of High Performance Multicache Parallel Texture Mapping. In *Proceedings of the 12th International Conference on Supercomputing*, pages 289–296, July 1998.

[105] J. Walton. The AMD Trinity Review (A10-4600M): A New Hope. May 2012. Available at http://www.anandtech.com/show/5831/amd-trinity-review-a10-4600m-a-new-hope/.

[106] C-J. Wu et al. SHiP: Signature-Based Hit Predictor for High Performance Caching. In *Proceedings of the 44th International Symposium on Microarchitecture*, pages 430–441, December 2011.

[107] Y. Xie and G. H. Loh. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 174–183, June 2009.

[108] M. Yuffe et al. A Fully Integrated Multi-CPU, GPU, and Memory Controller 32 nm Processor. In *Proceedings of the International Solid-State Circuits Conference*, pages 264–266, February 2011.

[109] 2015 International Technology Roadmap for Semiconductors (ITRS). http://www.semiconductors.org.

[110] 3D Mark Benchmark. http://www.3dmark.com/.

Appendix Study on Application Working-set

This appendix presents LLC miss rates of the applications (3D rendering, CPU, and GPGPU) studied in this dissertation as a function of the LLC capacity. Figures A.1, A.2, A.3, and A.4 present the results for the 3D rendering applications. Figures A.5 and A.6 present results for CPU and GPGPU applications, respectively. As these results show, the cache sensitivities of graphics, CPU, and GPGPU appli-

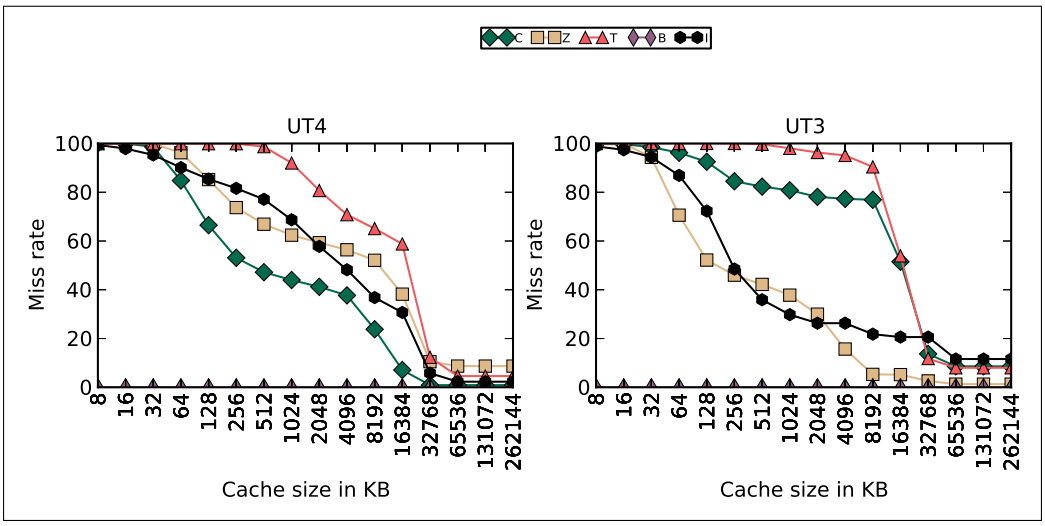


Figure A.1: Working sets of different LLC access streams

cations are significantly different. Compared to the GPU applications (both 3D

rendering and GPGPU), the CPU applications are more sensitive to cache capacity. For the 3D rendering applications, the sensitivity varies across different streams. The GPGPU applications show mixed behavior. On one hand, lbm, fastwalsh, and reduction have very large working sets, while bfs and cfd show higher sensitivity and their LLC miss rates drop quickly with increasing LLC capacity.

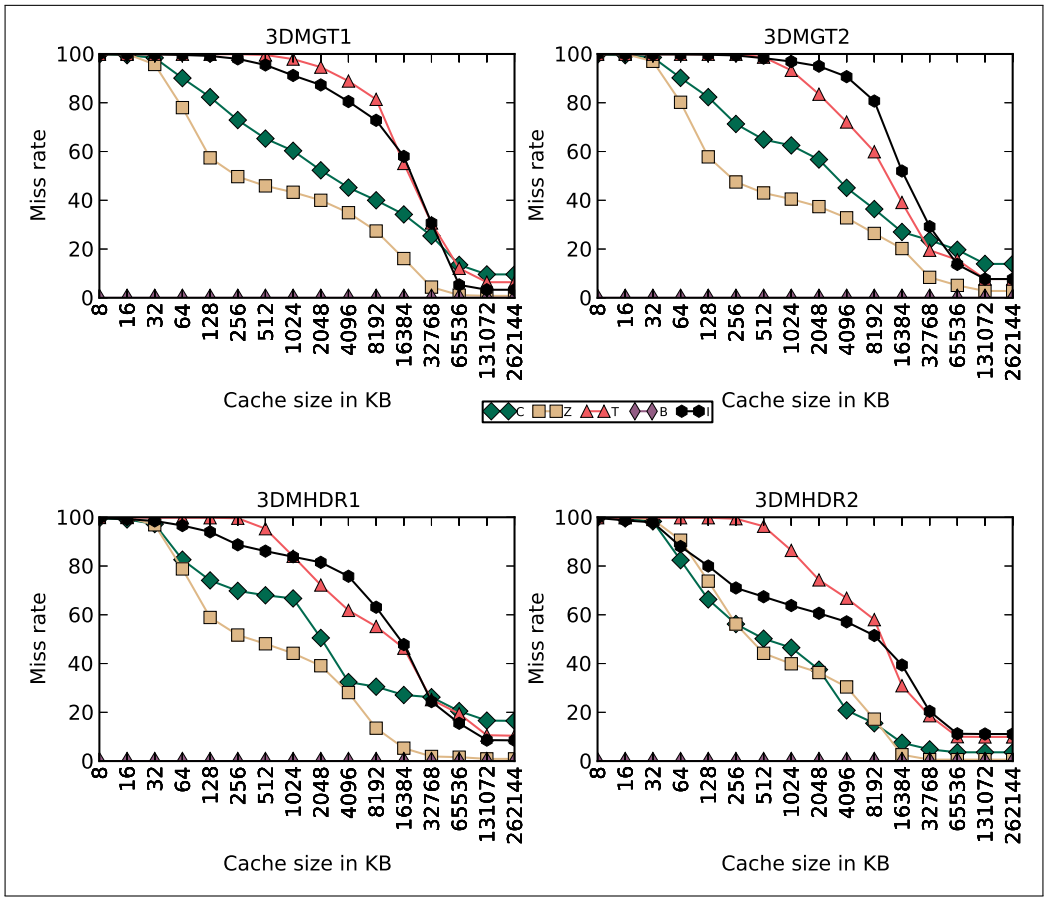


Figure A.2: Working sets of different LLC access streams

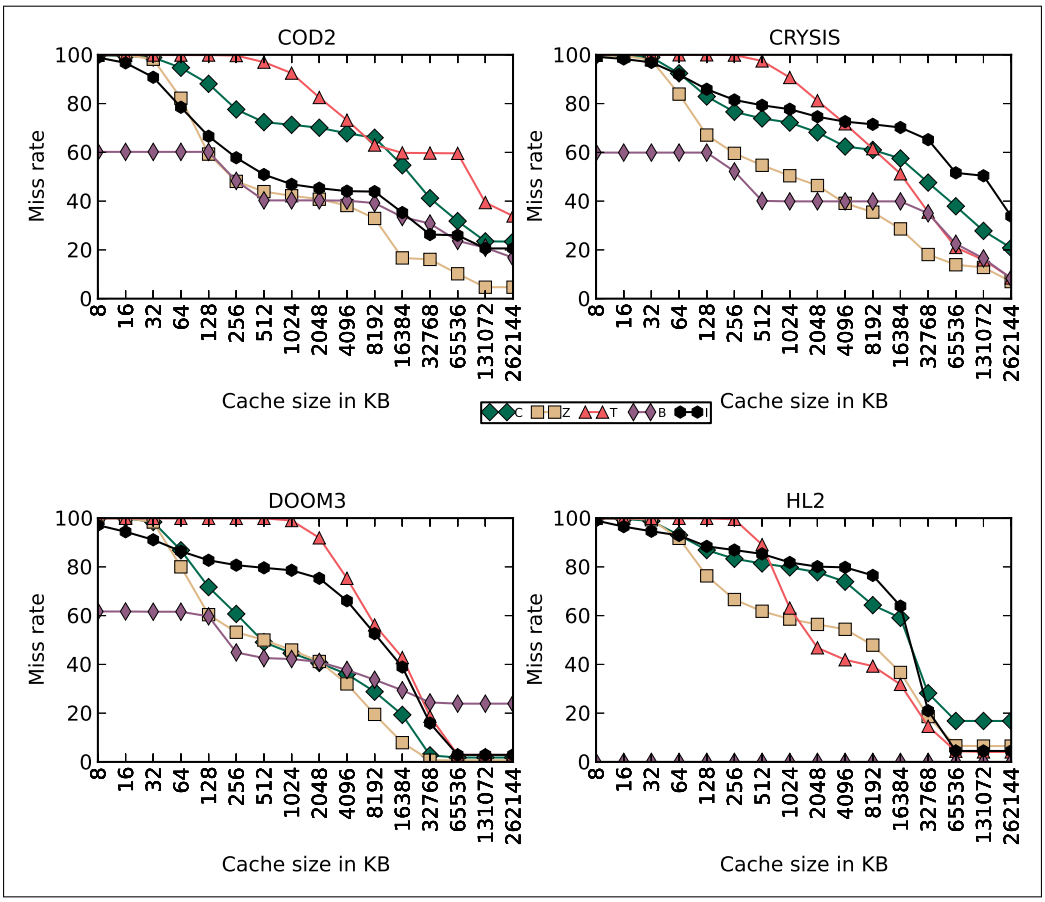


Figure A.3: Working sets of different LLC access streams

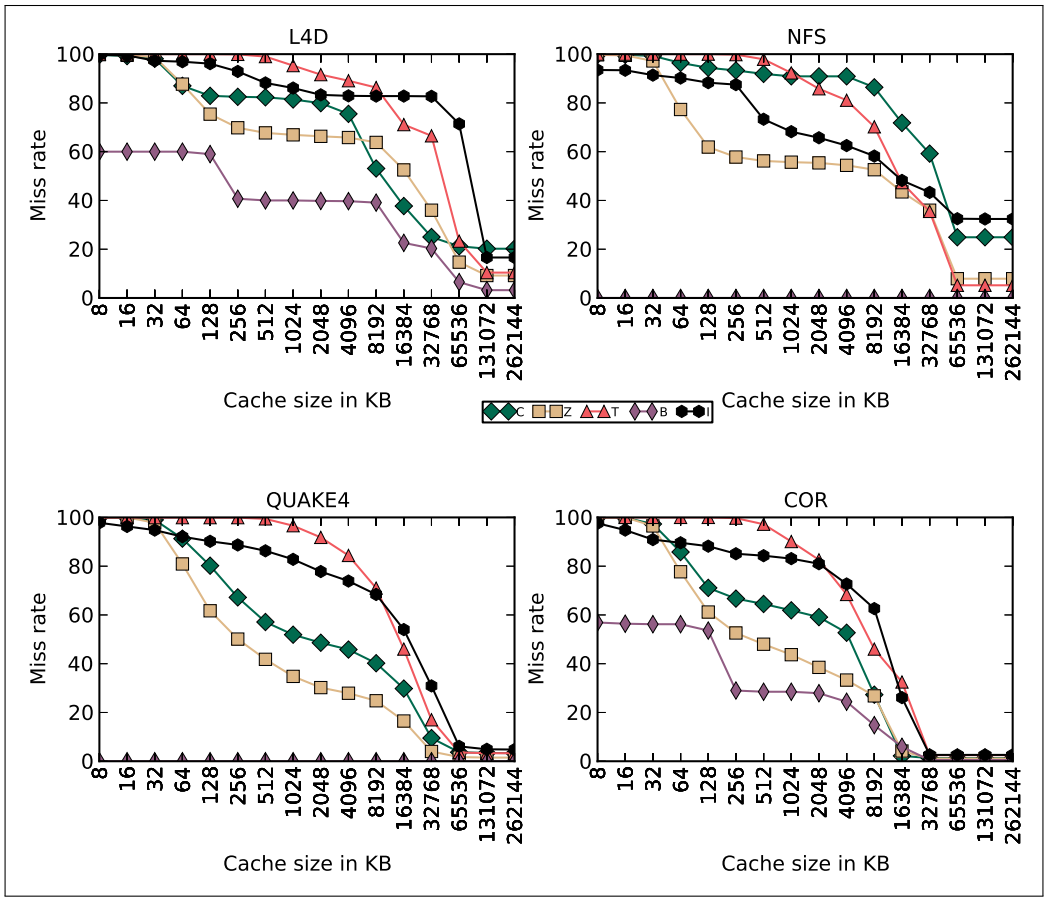


Figure A.4: Working sets of different LLC access streams

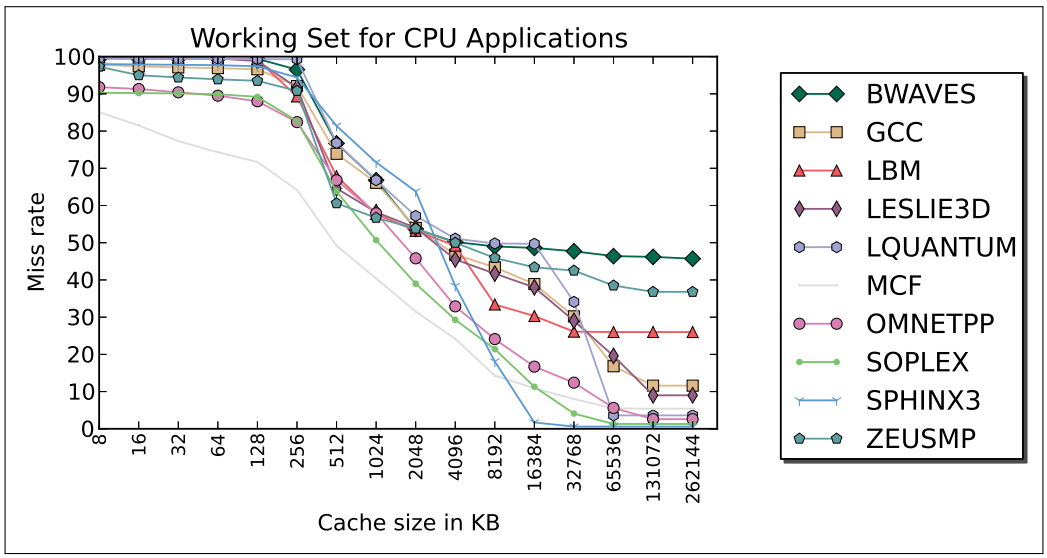


Figure A.5: Working sets of CPU applications

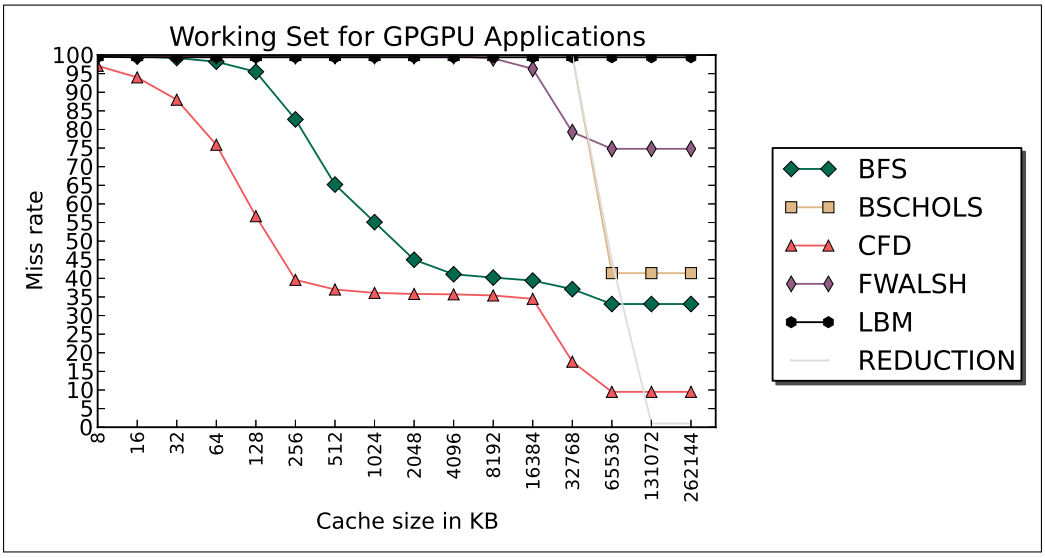


Figure A.6: Working sets of GPGPU applications

Appendix Publications

1. Siddharth Rai and Mainak Chaudhuri, Exploiting Dynamic Reuse Probability to Manage Shared Last-level Caches in CPU-GPU Heterogeneous Processors. *In Proceedings of the 30th ACM International Conference on Supercomputing*, article no. 3, June 2016.

2. Siddharth Rai and Mainak Chaudhuri, Improving CPU Performance through Dynamic GPU Access Throttling in CPU-GPU Heterogeneous Processors. *In Proceedings of the 26th IEEE International Heterogeneity in Computing Workshop*, pages 18-29, May 2017.

3. Siddharth Rai and Mainak Chaudhuri, Using Criticality of GPU Accesses in Memory Management for CPU-GPU Heterogeneous Multi-core Processors. *In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, October 2017.