Exploiting Dynamic Reuse Probability to Manage Shared Last-level Caches in CPU-GPU Heterogeneous Processors

Siddharth Rai
Dept. of Computer Science and Engineering
Indian Institute of Technology, Kanpur
Uttar Pradesh 208016, INDIA
sidrai@cse.iitk.ac.in

Mainak Chaudhuri
Dept. of Computer Science and Engineering
Indian Institute of Technology, Kanpur
Uttar Pradesh 208016, INDIA
mainakc@cse.iitk.ac.in

ABSTRACT

Recent commercial chip-multiprocessors (CMPs) have integrated CPU as well as GPU cores on the same die. In today's designs, these cores typically share parts of the memory system resources. However, since the CPU and the GPU cores have vastly different resource requirements, challenging resource partitioning problems arise in such heterogeneous CMPs. In one class of designs, the CPU and the GPU cores share the large on-die last-level SRAM cache. In this paper, we explore mechanisms to dynamically allocate the shared last-level cache space to the CPU and GPU applications in such designs. A CPU core executes an instruction progressively in a pipeline generating memory accesses (for instruction and data) only in a few pipeline stages. On the other hand, a 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 last-level cache management policies, the CPU and the GPU data streams can interfere with each other leading to significant loss in CPU and GPU performance accompanied by degradation in GPU-rendered 3D animation quality. Our proposal 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 coverage while requiring few tens of kilobytes of storage. We use the estimated reuse probabilities to design shared last-level cache policies for handling hits and misses to reads and writes from both types of cores. Studies on a detailed heterogeneous CMP simulator show that compared to a state-of-the-art baseline with a 16 MB shared last-level cache, 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% while improving the performance of the co-running quad-core CPU workload mixes by 7% on average and up to 19%.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICS '16, June 01-June 03, 2016, Istanbul, Turkey
© 2016 ACM. ISBN 978-1-4503-4361-9/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2925426.2926266

CCS Concepts

•Computer systems organization → Heterogeneous (hybrid) systems;

Keywords

CPU-GPU integration; shared last-level cache; temporal reuse

1. INTRODUCTION

Recent commercial chip-multiprocessors (CMPs) have integrated CPU as well as GPU cores on the same die. These include AMD's accelerated processing unit (APU) family [2, 30, 65] and Intel's Sandy Bridge, Ivy Bridge, Haswell, Broadwell, and Skylake processors [11, 18, 28, 29, 52, 61, 68]. In these processors, the CPU and the GPU cores share significant portions of the memory system resources. For example, in AMD APU architectures, the CPU and the GPU cores share everything beyond the on-die cache hierarchy including memory controllers and DRAM banks. In the Intel's integrated designs, the CPU and the GPU cores share the large on-die last-level (L3) cache in addition to sharing the in-package L4 eDRAM cache (available in Haswell, Broadwell, and Skylake parts), the on-die interconnect, memory controllers, and the DRAM banks. Such tight integration of CPU and GPU cores necessitates dynamic partitioning of the memory system resources so that both types of cores can be used simultaneously to obtain the best performance from such systems. In this paper, we squarely focus on the designs where the last level of the cache hierarchy is shared between the CPU and the GPU cores and address the problem of dynamically allocating the shared cache space to the CPU and the GPU 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 buffers (Z buffers holding pixel depth values), hierarchical depth buffers (HiZ buffers holding hierarchical depth values to reduce Z buffer bandwidth [16]), render targets (holding pixel color data), texture maps, 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 a GPGPU application) exercises only a portion of the rendering pipeline (primarily the shader cores) and does not need to access all these data types.

Ideally, a heterogeneous CMP makes simultaneous use of both CPU and GPU cores. As the CPU and the GPU working sets contend for the last-level cache (LLC) capac-

ity, the destructive interference arising from this contention can significantly hamper the progress of the tasks being executed by the CPU and the GPU cores. More 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 heterogeneous mode. Figure 1 shows the speedup achieved by the standalone execution over the heterogeneous execution when the simulated CMP is equipped with a shared 16 MB LLC.[1] The top panel shows the results for a single CPU core and single GPU configuration (1C1G), the middle panel shows the results for a dual-core CPU and single GPU configuration (2C1G), and the bottom panel shows the results for a quad-core CPU and single GPU configuration (4C1G). In all cases, the GPU executes 3D scene rendering jobs drawn from fourteen DirectX 9 and OpenGL game titles and four general-purpose CUDA applications. The corresponding eighteen single-, dual-, and quad-core CPU jobs are prepared by randomly mixing memory-sensitive SPEC 2006 applications. The heterogeneous workloads are denoted by S1-S18, D1-D18, and Q1-Q18, respectively for 1C1G, 2C1G, and 4C1G configurations. The set of eighteen GPU workloads is kept fixed in these configurations. In the first fourteen heterogeneous mixes, the GPU workloads are 3D scene rendering jobs, while in the last four heterogeneous mixes, the GPU workloads are general-purpose CUDA applications. The CPU (GPU) bar shows the speedup enjoyed by the CPU (GPU) job when it runs alone compared to when it runs together with a GPU (CPU) job.

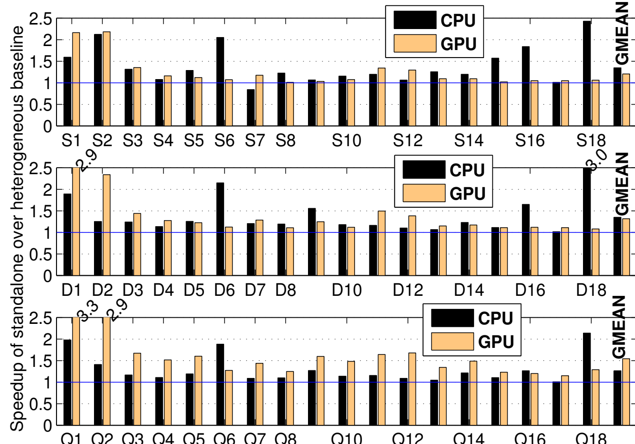

Figure 1: Speedup achieved by standalone execution over heterogeneous execution with a 16 MB LLC.

From these results, we see that for a 1C1G CMP, if only the CPU is active, the CPU job enjoys an average (geometric mean) speedup of 35% (see the CPU bar in the GMEAN group of the top panel) compared to the situation when the CPU job runs along with a GPU workload. Similarly, if only the GPU is active, the GPU job experiences an average speedup of 21% compared to running together with a CPU job.[2] As the number of active CPU cores increases, the interference experienced by the GPU workloads increases sharply. For a 2C1G CMP, a standalone GPU job enjoys a 32% average speedup compared to when it has a co-running dual-core CPU workload. A standalone dual-core CPU workload enjoys a 35% speedup on average compared to when it has a co-running GPU workload. For a 4C1G CMP, the speedup figures observed by the standalone GPU and CPU runs are 54% and 26%, respectively. As expected, with increasing CPU core count, the GPU workloads suffer more compared to the CPU workloads in the heterogeneous mode. A prior study exploring GPU concurrency management in CPU-GPU heterogeneous processors also examined similar performance interference between co-running CPU and GPGPU applications [31].

These results clearly indicate that without a carefully designed LLC management policy, the performance degradation in both CPU and GPU workloads can be significant in heterogeneous CMPs. More importantly, the large degradation in graphics frame rates can severely hurt the visual experience of the end-users. In this work, we attempt to recover some of this lost performance by designing an LLC policy that takes into account dynamic reuse probabilities of different GPU as well as CPU data streams. We motivate our proposal by quantifying upper bounds on performance benefits as well as LLC miss savings that the CPU and the GPU workloads can enjoy through better management of the shared LLC in heterogeneous CMPs (Section 4). 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 (Section 5). Our detailed simulation results show that the proposed algorithm is able to improve the GPU performance by 12% on average (up to 51%) and CPU workload performance by 7% on average (up to 19%) compared to a state-of-the-art heterogeneous baseline CMP in the 4C1G configuration (Section 6).

2. RELATED WORK

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

2.1 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 [53]. 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 [25]. A decision-tree based insertion age inference algorithm has also been proposed [37].

The concepts of re-reference prediction value (RRPV) and re-reference interval prediction (RRIP) are introduced in [24]. 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)

[1] Section 3 discusses our simulation environment.

[2] The speedup figures experienced by the standalone runs can also be interpreted as the slowdown suffered by the heterogeneous runs.

applies the technique proposed in [25] 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 [66]. 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 [33], estimation of approximate next-use distance [48], and estimation of protection distance [13] 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 [35]. 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 [34]. 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 [59].

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 [21, 36, 38, 39, 41, 45]. 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 [5]. 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 [4, 15].

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 [54]. The promotion/insertion pseudo-partitioning (PIPP) policy improves UCP by designing smart insertion and promotion policies for cache blocks within each partition [67]. Subsequent proposals such as Vantage [58] and PriSM [47] 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 [62]. 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 [14]). 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.

2.2 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 [43]. 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 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 [50]. 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. We study the potential improvements that can come from LLC bypassing in Section 4 and quantitatively compare our proposal with HeLM in Section 6.

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.

2.3 LLC Management in Discrete GPUs

The graphics stream-aware probabilistic caching proposal discusses algorithms for improving the LLC performance in discrete GPUs [14]. 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 [8], shader cores' L1 cache allocation policies based on a certain priority assignment to the shader threads executing GPGPU-style scientific computing workloads [44], and various optimization on the texture cache architecture [9, 10, 17, 22, 23, 64]. Additionally, there have been proposals exploring shader thread scheduling mechanisms that are shader cores' L1 cache performance-aware [26, 27, 32, 42, 55] or memory divergence-aware [56]. 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 [3]. 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. Further, in Section 4, we show that the shader data/instruction streams offer no opportunity for improving LLC misses in the 3D scene rendering applications.

3. SIMULATION ENVIRONMENT

We use an in-house modified version of the Multi2Sim simulator [63] to model the CPU cores of the simulated heterogeneous CMP. Each dynamically scheduled out-of-order issue x86 core is clocked at 4 GHz. 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 for 22 nm node[3]). All L1 and L2 caches have a block size of 64 bytes.

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 [51]. 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 sixteen 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.[4] 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 per 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 the 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. Each of these units can process one quad-pixel stamp every cycle leading to a peak pixel fill rate of 64 GPixels/second. The GPU has a three-level non-inclusive texture cache hierarchy resembling the texture cache hierarchy of Intel's integrated GPUs (Gen7 onward) [28]. 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 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 [40], which makes use of the GPUOcelot tool [12] for capturing the CUDA application instructions. Since the CUDA applications make use of the shader cores only, the GPU simulator contains a detailed model of the shader core island of the GPU. We borrow the following configuration of this GPU model from the recently published studies in the same area [43]. The GPU has six shader cores (similar to the streaming multiprocessors of the Nvidia GPUs), each clocked at 1.5 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 warps every fourth cycle. The peak execution throughput of each shader core is sixteen single-precision floating-point operations per cycle. Each shader core is equipped with a 4 KB eight-way instruction cache, a 32 KB 8-way data cache, an 8 KB texture cache, an 8 KB constant cache, and a 16 KB software-managed shared memory.

Depending on the type of the GPU workload being executed, one of the two GPU models gets attached to the rest of the heterogeneous CMP. The shared LLC of the heterogeneous CMP receives requests that miss in the CPU cores' L2 caches or GPU's vertex cache, HiZ cache, shader caches, L2 texture cache, L2 depth cache, or L2 color cache (we consider all requests coming from the shader cores as originating from the shader cache misses). The LLC is 16 MB 16-way set-associative with a lookup latency of ten cycles. The LLC maintains inclusion for all CPU data and instructions. 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 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 [57]. Each DRAM module is eight-way banked single-rank DDR3-2133 with 14-14-14 latency parameters and burst length eight. The memory controllers implement the FR-FCFS scheduling algorithm. The CPU cores along with

[3] We use the CACTI distribution that comes with the McPAT tool [20].

[4] To correspond to the usual terminology, each thread (or sixteen-way vector thread) here can be seen as a warp or wavefront issuing sixteen operations in lock-step in a cycle.

Table 1: Graphics frame details

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

Table 2: CUDA application details

Application	Source // Input // Executed portion // Thread and block configuration
cfd (euler3d)	Rodinia 3.0 [6, 7] // fvcorr.comn.097k // First 71 kernel invocations // 759 blocks×128 threads/block
blackscholes	CUDA SDK 4.2 // Generated by app. // Full // 480 blocks×128 threads/block
fastwalsh	CUDA SDK 4.2 // Generated by app. // First two kernel invocations // 8192 blocks×256 threads/block
reduction	CUDA SDK 4.2 // 16K elements // Sixth kernel // 64 blocks×256 threads/block

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 heterogeneous workloads used in this study are built by mixing CPU applications drawn from the SPEC 2006 suite and 3D scene rendering jobs drawn from fourteen popular DirectX 9 and OpenGL game titles as well as four CUDA applications drawn from publicly available benchmark suites. The DirectX and OpenGL API traces for the selected 3D animation frames are obtained from the Attila simulator distribution and the 3DMark06 suite [69]. The simulated game regions (i.e., sequences of frames) are selected at random after skipping over the initial sequence and detailed in Table 1. The details of the selected CUDA applications are shown in Table 2. 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 2006 applications are simulated in execution-driven mode on the CPU cores. Table 3 lists the 1C1G, 2C1G, and 4C1G workload mixes (S1-S18, D1-D18, and Q1-Q18, respectively) 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 [60] and early-finishing applications continue 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 of the CPU mixes is measured in terms of average instructions per cycle throughput. The GPU performance for the 3D scene rendering jobs is measured in terms of average frame rate and for the CUDA applications, the number of execution cycles to complete the job is used.

4. MOTIVATION

In this section, we motivate our proposal by validating that saving LLC misses can bring performance improvements in the CPU-GPU heterogeneous system under consideration. First, we show that there is a significant room left for saving LLC misses compared to the state-of-the-art proposals. Second, we quantify the potential improvements in the GPU performance as the LLC gradually approaches an ideal cache that only suffers from the compulsory misses. This validation is necessary before embarking on our proposal because the GPU is designed to have reasonable latency tolerance and as a result, LLC miss savings may not always translate to performance improvements. The same aspect of the GPU architecture motivates our third study that explores the potential impact on the performance of the CPU and the GPU when the GPU read misses are forced to selectively bypass the LLC and not allocate in the LLC.

4.1 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 [1, 49]. We first briefly discuss the evaluated proposals in the following.

Baseline: The baseline LLC follows the originally proposed SRRIP algorithm [24] 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 [46]. Such texture data are usually referred to as dynamic texture data [19]. 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 con-

Table 3: Heterogeneous workload mixes

GPU workload	CPU workload mix (Sn: 1C1G // Dn: 2C1G // Qn: 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

tents can also be reused by the texture sampler for rendering shadow [19].

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 [24, 25, 53]. Finally, 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: The DRRIP [24] and TADRRIP [24] policies were introduced in Section 2. The TADRRIP policy treats the CPU cores and the GPU as different independent threads and lets each thread choose the best insertion RRPV (among two and three) for read misses. The write miss and write hit policies are same as the baseline.

SHiP-mem: The SHiP-mem [66] policy was introduced in Section 2. 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. Recall that 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 [66] 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 that miss in the LLC. 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.).[5] 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 as the baseline.

OPT, OPT+Bypass, Baseline+Bypass: The OPT policy implements Belady's MIN replacement algorithm [1, 49] 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

[5] This is also the reason why it is not possible to have an effective GPU implementation of the other existing proposals (such as SDBP [36]) that rely on the program counters associated with the LLC accesses.

collecting an LLC access trace for each heterogeneous workload mix and running 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.

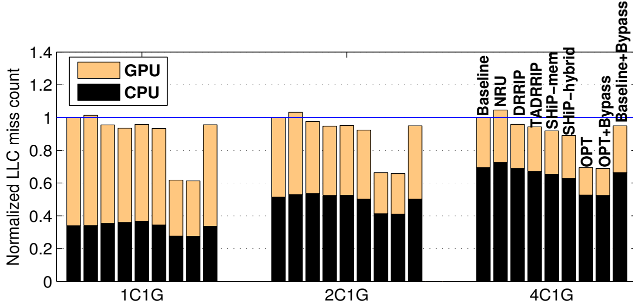

Figure 2: Normalized average read miss count.

Figure 2 shows the normalized number of LLC read misses for the LLC policies averaged over the 1C1G, 2C1G, and 4C1G heterogeneous workloads. 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 configurations, 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 improve 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 section.

4.2 GPU Performance with Ideal LLC

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 shows the progressive speedup (in terms of frame rate) observed by the 3D rendering applications as color (C),

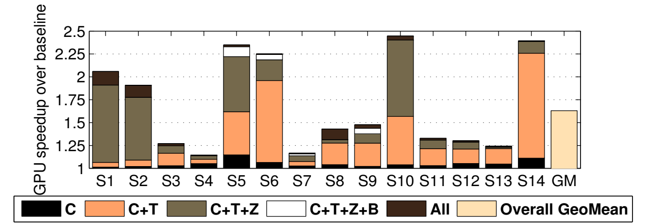

Figure 3: Potential improvement in frame rate.

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 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 remaining GPU streams. 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.

Table 4: Speedup of CUDA applications with ideal LLC

S15	S16	S17	S18
1.22	1.12	1.26	2.82

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 4 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.

4.3 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 2). 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 4 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 mem-

ory 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 [50]. We attribute this difference to a wider variety of GPU applications considered in our study.

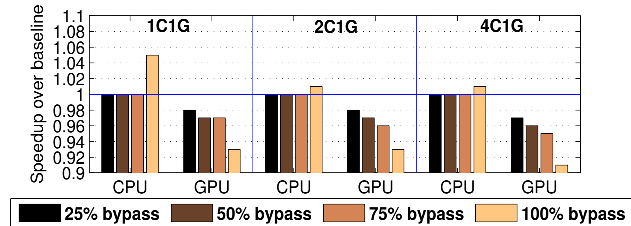

Figure 4: Performance speedup with random GPU read miss bypass for a 16 MB LLC.

5. DYNAMIC REUSE PROBABILITY IN LLC MANAGEMENT

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.

5.1 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. The WSS cache is a traditional set-associative cache. 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 5 shows a typical WSS cache entry.

V	TAG	V0	W0	SID0	● ● ●	V3	W3	SID3

Figure 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 kth 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. Figure 6 shows the flow for looking up the WSS cache on an access from stream S for a block which is the mth 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 all the reuse and access counters are halved so that we can capture phase changes. Next, we present the proposed sub-policies.

5.2 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

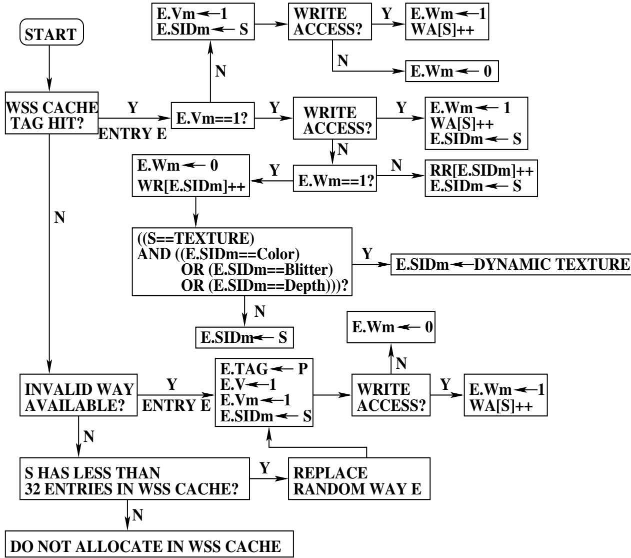

Figure 6: Access/Update protocol flow for the WSS cache and the accompanying reuse and access counters. Vm, Wm, and SIDm correspond respectively to the V, W, and stream id of the mth tracked block. The MAX_REUSE register is not shown.

read miss policy by borrowing from the vast body of existing research that deals with LLC insertion policies on read misses discussed in Section 2 and evaluated in Section 4. 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 [24], 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 7 presents a high-level depiction of the proposed read miss policy.

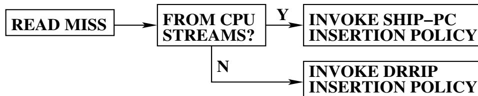

Figure 7: Read miss policy.

5.3 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.

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. 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 8 summarizes our write miss policy proposal.

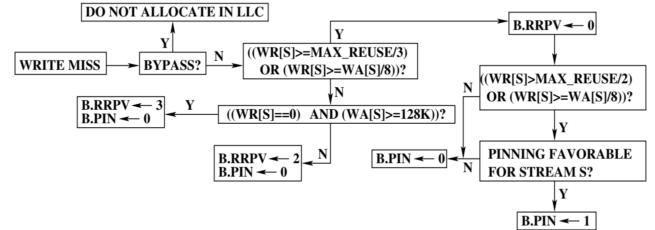

Figure 8: Write miss policy for a block B coming from stream S.

5.4 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 9 shows the congestion-oblivious and congestion-aware write hit policies.

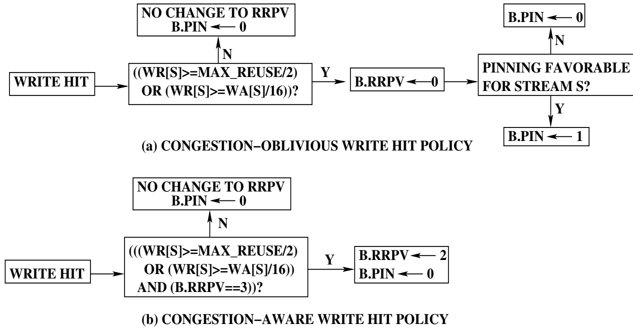

Figure 9: Write hit policies for a block B coming from stream S.

5.5 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 10 summarizes our read hit policy proposal.

5.6 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

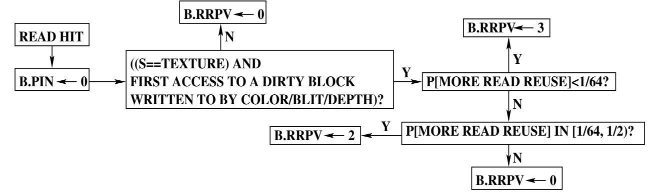

Figure 10: Read hit policy for a block B coming from stream S. P[E] denotes the probability of event E.

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.

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 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.

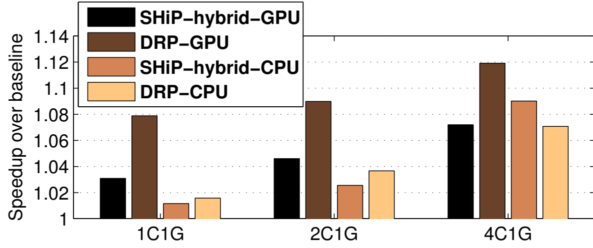

Figure 11: Average speedup comparison.

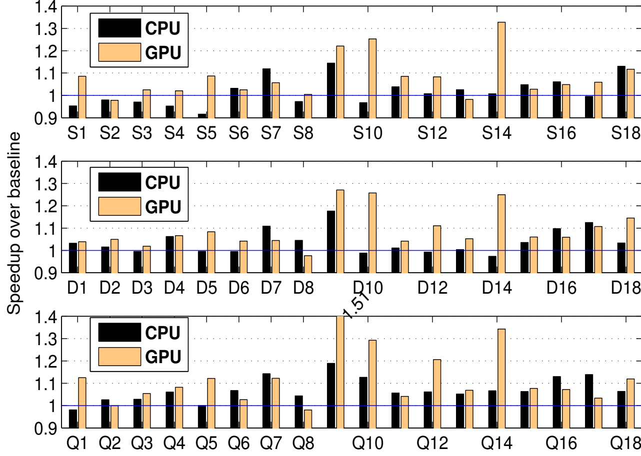

Figure 12: Speedup achieved by the DRP-aware proposal for the mixes.

Figure 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% improvement 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.

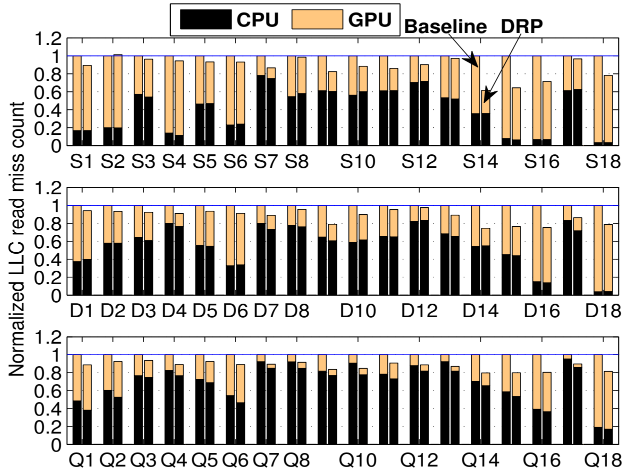

Figure 13: Normalized read miss count of the mixes.

To understand the source of the performance improvements, Figure 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 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 improvement in the volume of CPU misses. Additionally, D4 and D9 show some improvement 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 improvement 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 improve 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 improve 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.

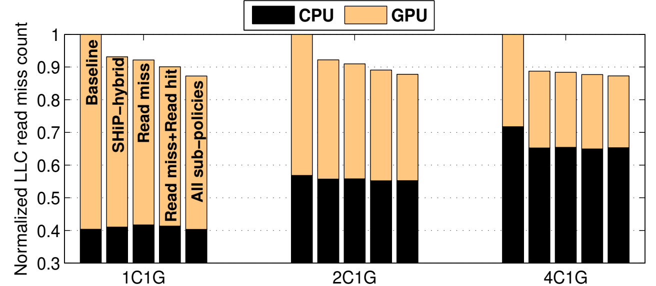

Figure 14: Normalized average read miss count.

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 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 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 improvement 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 11.

6.1 Comparison to Related Proposals

There have been two proposals, namely, TAP [43] and HeLM [50], for managing the shared LLC in heterogeneous CMPs. These proposals were briefly introduced in Section 2. 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.

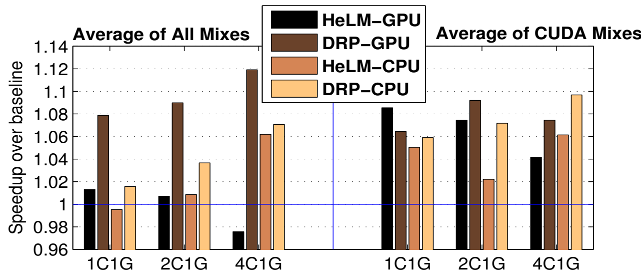

Figure 15: Speedup of HeLM and DRP.

Figure 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 4). 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 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.

7. SUMMARY

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.

8. REFERENCES

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

[2] 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.

[3] 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.

[4] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman. 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.

[5] 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.

[6] 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.

[7] 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.

[8] X. Chen, L-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W-M. Hwu. Adaptive Cache Management for Energy-efficient GPU Computing. In *Proceedings of the 47th International Symposium on Microarchitecture*, pages 343–355, December 2014.

[9] C. J. Choi, G. H. Park, J. H. Lee, W. C. Park, and T. D. Han. 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.

[10] 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.

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

[12] 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.

[13] N. Doung, D. Zhao, T. Kim, R. Cammarato, M. Valero, and A. V. Veidenbaum. Improving Cache Management Policies Using Dynamic Reuse Distances. In *Proceedings of the 45th International Symposium on Microarchitecture*, pages 389–400, December 2012.

[14] J. Gaur, R. Srinivasan, S. Subramoney, and M. Chaudhuri. 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.

[15] 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.

[16] 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.

[17] 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.

[18] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, J. Hong, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The Fourth Generation Intel Core Processor. In *IEEE Micro*, **34**(2):6–20, March/April 2014.

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

[20] 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/.

[21] 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.

[22] 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.

[23] 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.

[24] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. 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.

[25] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr., and J. Emer. 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.

[26] 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.

[27] 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.

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

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

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] S. Khan, D. A. Jimènez, D. Burger, and B. Falsafi. 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.

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

[40] 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/.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] H. Liu, M. Ferdman, J. Huh, and D. Burger. 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.

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

[47] 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.

[48] 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.

[49] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. In *IBM Systems Journal*, **9**(2): 78–117, 1970.

[50] V. Mekkat, A. Holey, P-C. Yew, and A. Zhai. 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.

[51] V. Moya, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa. 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.

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

[53] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 381–391, June 2007.

[54] 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.

[55] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 45th International Symposium on Microarchitecture*, pages 72–83, December 2012.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

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

[63] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. 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.

[64] 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.

[65] 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/.

[66] C-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr., and J. Emer. SHiP: Signature-Based Hit Predictor for High Performance Caching. In *Proceedings of the 44th International Symposium on Microarchitecture*, pages 430–441, December 2011.

[67] 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.

[68] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. 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.

[69] 3D Mark Benchmark. http://www.3dmark.com/.