Using Criticality of GPU Accesses in Memory Management for CPU-GPU Heterogeneous Multi-Core Processors

SIDDHARTH RAI and MAINAK CHAUDHURI, Indian Institute of Technology Kanpur

Heterogeneous chip-multiprocessors with CPU and GPU integrated on the same die allow sharing of critical memory system resources among the CPU and GPU applications. Such architectures give rise to challenging resource scheduling problems. In this paper, we explore memory access scheduling algorithms driven by criticality of GPU accesses in such systems. Different GPU access streams originate from different parts of the GPU rendering pipeline, which behaves very differently from 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 scene 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 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. Extensions proposed for the mixes containing GPGPU applications, which do not have any quality of service requirement, improve the performance by 7% on average for these mixes.

CCS Concepts: • **Computer systems organization** → *Heterogeneous (hybrid) systems*;

Additional Key Words and Phrases: CPU-GPU heterogeneous multi-core, GPU access criticality, DRAM access scheduling, 3D rendering, GPGPU

ACM Reference format:
Siddharth Rai and Mainak Chaudhuri. 2017. Using Criticality of GPU Accesses in Memory Management for CPU-GPU Heterogeneous Multi-Core Processors. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 133 (September 2017), 23 pages.
https://doi.org/10.1145/3126540

1 INTRODUCTION

A heterogeneous chip-multiprocessor (CMP) makes simultaneous use of both CPU and GPU cores. Such architectures include AMD's accelerated processing unit (APU) family [3, 26, 65] and Intel's Sandy Bridge, Ivy Bridge, Haswell, Broadwell, and Skylake processors [8, 14, 24, 25, 48, 66]. In this study, we explore heterogeneous CMPs that allow the CPU cores and the GPU to share the on-die interconnect, the last-level cache (LLC), the memory controllers, and the DRAM banks, as found in the Intel's product line of integrated GPUs. The GPU workloads are of two types: massively parallel computation exercising only the shader cores (GPGPU or GPU computing) and

This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2017 and appears as part of the ESWEEK-TECS special issue.
Authors' addresses: S. Rai and M. Chaudhuri, Department of Computer Science and Engineering, Indian Institute of Technology Kanpur, Uttar Pradesh 208016, India; emails: {sidrai, mainakc}@cse.iitk.ac.in.

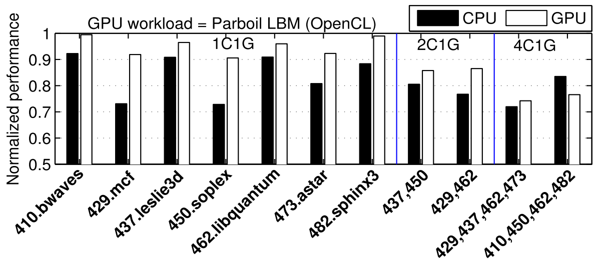

Fig. 1. Performance of heterogeneous mixes relative to standalone performance on Core i7-4770.

multi-frame 3D animation utilizing the entire rendering pipeline of the GPU. Each of these two types of GPU workloads can be co-executed with general-purpose CPU workloads in a heterogeneous CMP. These heterogeneous computational scenarios are seen in embedded, desktop, workstation, and high-end computing platforms employing CPU-GPU MPSoCs. In this paper, we model such heterogeneous computational scenarios by simultaneously scheduling SPEC CPU workloads on the CPU cores and 3D scene rendering or GPGPU applications on the GPU of a heterogeneous CMP. In these computational scenarios, the working sets of the jobs co-scheduled on the CPU and the GPU cores contend for the shared LLC capacity and the DRAM bandwidth causing destructive interference.

To understand the extent of the CPU-GPU interference, we conduct a set of experiments on an Intel Core i7-4770 processor (Haswell)-based platform. This heterogeneous processor has four CPU cores and an integrated HD 4600 GPU sharing an 8 MB LLC and a dual-channel DDR3-1600 16 GB DRAM (25.6 GB/s peak DRAM bandwidth) [17]. We prepare eleven heterogeneous mixes for these experiments. Every mix has LBM from the Parboil OpenCL suite [58] as the GPU workload using the long input (3000 time-steps). LBM serves as a representative for memory-intensive GPU workloads.[1] 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 shows, for each mix (identified by the constituent CPU workload on the x-axis), the performance of the CPU and the 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 these workloads run together, they contend for the shared memory system resources leading to severe performance degradation, As Figure 1 shows, the loss in CPU performance varies from 8% (410.bwaves in the 1C1G group) to 28% (the first mix in the 4C1G group). The GPU performance degradation ranges from 1% (the first mix in the 1C1G group) to 26% (the first mix in the 4C1G group). The GPU workload degrades more with the increasing number of active CPU cores. Similar levels of interference have been reported in simulation-based studies [2, 20, 27, 36, 39, 45, 49, 56, 57, 64]. In this paper, we attempt to recover some portion of the lost GPU performance by proposing a novel memory access scheduler driven by GPU performance feedback.

Prior proposals have studied specialized memory access schedulers for heterogeneous systems [2, 20, 45, 57, 64]. These proposals modulate the priority of all GPU or all CPU requests in

[1]Experiments done with a larger set of memory-intensive GPU workloads show similar trends.

bulk depending on the latency-, bandwidth-, and deadline-sensitivity of the current phase of the CPU and the GPU workloads. In this paper, we propose a new mechanism to dynamically identify a subset of GPU requests as critical and accelerate them by exercising fine-grain control over allocation of memory system resources. Our proposal is motivated by the key observation that the performance impact of accelerating different GPU accesses is not the same (Section 3). Since a GPU workload can exercise not only the programmable shader cores, but also numerous fixed function units such as texture samplers, early and late depth test units, color blenders, blitters, etc. with complex inter-dependence, dynamic identification of critical accesses in the GPUs requires novel insights and techniques compared to the existing criticality estimation techniques for CPU load instructions [12, 59]. For identifying the bottleneck GPU accesses, we model the information flow through the rendering pipeline of the GPU using a queuing network. We observe that the accesses to the memory system originating from the GPU shader cores and the fixed function units have complex inter-dependence, which is not addressed by the existing CPU load criticality estimation techniques. To the best of our knowledge, our proposal is the first to incorporate criticality information of fine-grain GPU accesses in the design of memory access schedulers for heterogeneous CMPs. While we focus on the criticality of GPU accesses only in this study, we note that it may be possible to incorporate the existing CPU criticality-based policies [12, 59] on top of our proposal to further improve the CPU performance.

3D animation is an important GPU workload. For these workloads, it is sufficient to deliver a minimum acceptable frame rate (usually thirty frames per second) due to the persistence of human vision; it is a wastage of resources to improve the GPU performance beyond the required level. Our proposal includes a highly accurate architecture-independent technique to estimate the frame rate of a 3D rendering job at run-time. This estimation algorithm does not require any profile pass and does not assume anything about the supported rendering algorithm. It works for immediate-mode rendering (IMR) supported by most high-end GPUs as well as tile-based deferred rendering (TBDR) supported by most mobile GPUs. We use the estimated frame rate to identify the 3D scene rendering applications that fail to meet the target performance level. Our proposal employs the criticality information to speed up the GPU workload only if the estimated frame rate is below the target level. We suitably extend our proposal for the GPGPU applications, which do not have any minimum performance requirement. Section 4 discusses our proposal. We summarize our contributions in the following.

- We present a novel technique for identifying the critical memory accesses sourced by the GPU rendering pipeline.
- We present mechanisms based on accurate frame rate estimation to identify the critical phases of 3D animation.
- We propose DRAM access scheduling mechanisms to partition the DRAM bandwidth between the critical GPU accesses, non-critical GPU accesses, and CPU accesses.

Simulations done with a detailed model of a heterogeneous CMP (Section 5) having one GPU and four CPU cores running mixes of DirectX, OpenGL, and CPU applications show that our proposal successfully identifies and accelerates the performance-critical GPU accesses. It improves the GPU performance by 15% on average without degrading the CPU performance much (Section 6). For mixes of CUDA and CPU applications, system performance improves by 7% on average.

2 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 [7, 10–12, 16, 18, 29, 31, 43, 44, 46, 52, 60–62]. Profile-guided assignment of DRAM channels to application groups has also been proposed [41]. Criticality estimation of load instructions [59] and criticality-driven memory access schedulers [12] 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 [4], to accelerate the critical shader cores that do not have enough short-latency warps [23], and to design an appropriate mix of shortest-job-first and FR-FCFS with the goal of accelerating the less latency-tolerant shader cores [33]. There have been studies on warp and thread block schedulers for improving the memory system performance [1, 21, 22, 28, 34, 35]. In contrast, our proposal addresses the criticality of accesses sourced by not only the shader core load instructions, but also numerous fixed function units in the GPU rendering pipeline.

Several studies have explored specialized memory access schedulers for heterogeneous systems [2, 20, 45, 57, 64]. The staged memory scheduler (SMS) clubs the memory requests from each source (CPU or GPU) into source-specific batches based on DRAM row locality [2]. 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 [20] proposed for mobile heterogeneous platforms employs dynamic progress estimation of tile-based deferred rendering (TBDR) [47, 51] 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 progress estimation algorithm is designed specifically for the GPUs employing TBDR, typically supported only in mobile GPUs such as ARM Mali [47], Kyro, Kyro II, and PowerVR from Imagination Technologies, and Imageon 2380, Xenos, Z430, and Z460 from AMD [51]. The dynamic priority scheduler study, however, shows the inefficiency of a previously proposed static priority scheduler that always offers higher priority to the CPU accesses [57]. The subsequently proposed deadline-aware memory scheduler for heterogeneous systems (DASH) further improves the dynamic priority scheme by incorporating three optimizations: (i) always offering higher priority to a GPU application that lags behind the target, (ii) offering the highest priority to short-deadline applications, and (iii) distinguishing between latency-sensitive and bandwidth-sensitive CPU workloads [64]. To estimate the dynamic progress of a GPU workload, the DASH proposal requires a priori knowledge about the number of memory requests that the workload would generate. Obtaining such information would typically require a profile pass. 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 [45]. Although such a proposal has the advantage of striping the two datasets differently across the banks of the two memory controllers, a subsequent study has shown that such static partitioning of memory resources can be sub-optimal [27]. Our proposal, in contrast, employs fine-grain criticality information regarding GPU accesses to design memory access management mechanisms.

Insertion and replacement policies to manage the shared LLC in the heterogeneous CMPs have been explored [36, 49]. Selective LLC bypass policies for GPU misses arising from latency-tolerant shader cores have been proposed with the goal of freeing up shared LLC space for CPU workloads and the non-bypassed (possibly critical) GPU shader core accesses [39].

3 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,

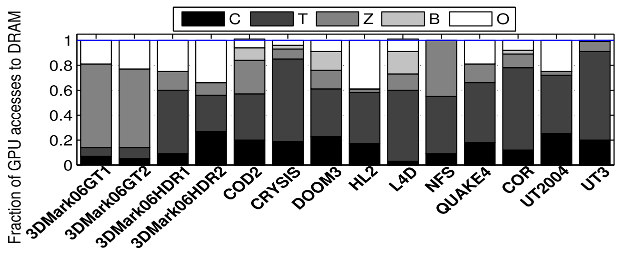

Fig. 2. Distribution of DRAM accesses from 3D scene rendering workloads.

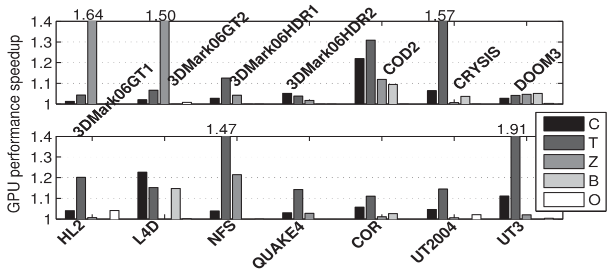

Fig. 3. Speedup achieved when each individual stream is made to behave ideally.

texture sampler data, pixel depth data, hierarchical depth data [13], 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.

Figure 2 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.[2] We consider the following stream categories: color (C), texture sampler (T), depth (Z), blitter (B), and everything else clubbed into the "other" (O) category.[3] Figure 2 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 3 evaluates the performance-criticality of each stream by making all accesses from that stream behave ideally in the memory system. More specifically, a stream is made to behave ideally by treating all its non-compulsory LLC misses as hits. This only means that instead of charging the miss latency for a non-compulsory miss to this stream, we charge the hit latency, do not send the request to DRAM, but execute the LLC replacement policy, as usual. Also, the treatment to all accesses from other streams is left unchanged. Figure 3 quantifies the speedup achieved by accelerating each stream in this way. The speedup is measured as the ratio of frame rates with

[2]Our simulation infra-structure is discussed in Section 5.

[3]The blitter unit is not used by all applications. This is a special fixed function unit used to copy color data from render target of DirectX or renderbuffer of OpenGL to a separate memory region and process the copied data before it can be sampled by the texture sampler. This is one possible way of doing "render-to-texture" or dynamic texturing, a well-known technique for generating photo-realistic dynamic texture maps [15]. Render-to-texture can be implemented without blitting as well [37].

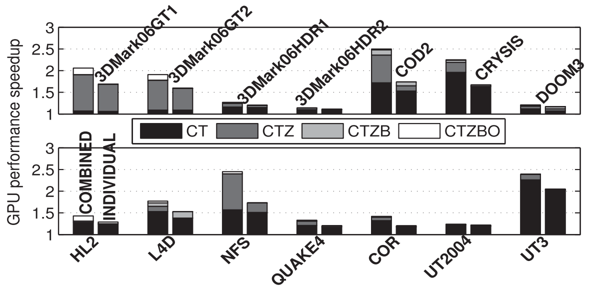

Fig. 4. Speedup achieved when a set of streams is made to behave ideally.

and without this optimization. For different applications, each stream shows different levels of performance-sensitivity. Different streams exploit the latency hiding capability of the GPU by different amounts leading to varying impacts on the critical path through the application. A comparison of Figures 2 and 3 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 (Figure 2), but accelerating the color stream brings much higher speedup compared to accelerating the depth stream (Figure 3). 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 4 quantifies the performance-criticality of a set of streams by treating all their non-compulsory LLC misses as hits. While several possible groupings of the streams are possible, 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". For example, the CT segment of the "INDIVIDUAL" bar shows the speedup of accelerating the color stream alone plus the speedup of accelerating the texture stream alone. On the other hand, the CT segment of the "COMBINED" bar shows the speedup of accelerating the color and the texture streams simultaneously. We observe that the combined speedup is much higher than the accumulated individual speedup in several applications. In some of the applications (e.g., 3DMark06GT1, 3DMark06GT2, COD2, CRYSIS, L4D, NFS, COR, and UT3), the gap between the heights of the two bars is significant. This indicates that there are certain inter-stream performance-dependencies that must be accelerated together. Some of these dependencies arise from known semantic data-flow rules, while others arise from the structure of the 3D rendering pipeline. 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 [15, 37]. 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.

For the GPGPU applications, the shader accesses constitute the dominant stream. To better understand the performance-sensitivity of the shader accesses in these applications, we further partition the shader data stream. In this paper, 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 [32, 38, 42]. More specifically, in each shader core

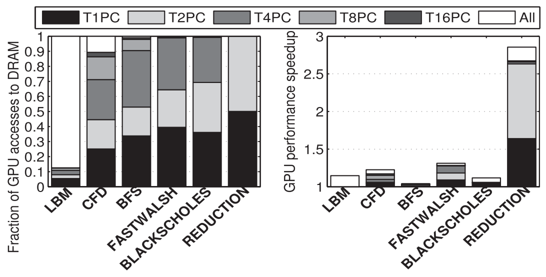

Fig. 5. Left: distribution of DRAM accesses from GPGPU workloads. Right: speedup achieved when the top n PC streams are made to behave ideally.

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 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. These data are collected on a simulated heterogeneous CMP with one CPU core and one GPU. Top four shader instructions can cover almost all DRAM accesses except for LBM and CFD. Only LBM and CFD require a larger number of shader instructions to have a reasonable coverage of the DRAM accesses.

The right panel of Figure 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. Except BFS, all applications improve by more than 10% when all shader accesses are treated ideally.

4 GPU ACCESS CRITICALITY

In this section, we present our proposal on identifying and managing critical GPU accesses in heterogeneous CMPs. Section 4.1 discusses the mechanism for identifying the critical GPU access streams. The mechanism for estimating the projected frame rate of 3D scene rendering applications is discussed in Section 4.2. The DRAM access scheduler presented in Section 4.3 finally shapes the priorities assigned to the CPU and GPU memory accesses.

4.1 Identifying Critical GPU Accesses

In the following, we present the mechanisms for selecting the critical accesses in 3D rendering and GPGPU workloads.

4.1.1 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.[4] The ZS unit removes the hidden surfaces

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

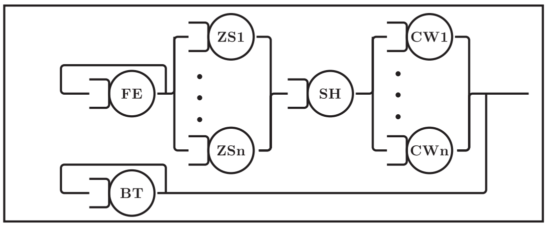

Fig. 6. Queuing network for GPU pipeline.

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. One ZS unit and one CW unit constitute one render output pipeline (ROP). The ZS and the CW units differ in the type of the processed data and the ALU functions. If the depth/stencil test is done before pixel shading, it is known as early-Z. In certain situations, the ZS unit may have to be invoked after SH and before CW. This is known as late-Z. Our proposal periodically classifies each unit as having high/low request rate and high/low throughput. This classification is used to identify the critical streams.

In the following, we first present a queuing network model for the 3D scene rendering pipeline. Our model captures all inter-unit interactions that take place when the GPU executes a 3D scene rendering job. This model is used by the critical stream selection algorithm to decide the order in which the units are considered for obtaining the criticality information. Next, we outline the per-unit thresholds used for high/low request rate and throughput classification. Finally, we present the algorithm used for selecting the critical GPU streams.

Queuing Model for Rendering Pipeline. We model the inter-dependence between the 3D rendering pipeline units using a queuing network shown in Figure 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 flows through FE, ZS, SH, and CW in that order leading to the output. This path gets activated during a draw operation when the input geometry is rendered to a target buffer (assuming early-Z). The second path, which connects BT to the output, gets activated during the blitting process. In this path, each request goes through multiple read/write operations before reaching the output.

Request Flow Monitoring. We monitor the request arrival and completion rates at each of the units to first identify the bottleneck units. Identification of the bottleneck unit in the 3D scene rendering pipeline is important for the selection of the critical GPU access streams. To find the bottleneck unit of the pipeline, we measure the request arrival and completion rates at each of the units shown in the model. 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 $w = 8$).[5] 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.

[5] All algorithm parameters are tuned meticulously.

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[6] 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 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.

4.1.2 GPGPU Workloads. For the GPGPU workloads, we employ a two-level algorithm invoked periodically for identifying the critical accesses. These workloads exercise only the shader cores of the GPU pipeline. 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, 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 3 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 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. HeLM, a mechanism for managing the shared LLC for heterogeneous processors, bypasses LLC misses from the latency-tolerant shader cores employing a complex algorithm for estimating the latency-tolerance of the cores [39]. In Section 6, we compare our proposal against the HeLM proposal.

4.2 Estimating Projected Frame Rate

The critical GPU accesses in a 3D scene rendering application are marked critical by our proposal only if the projected frame rate is below the target. Such projections need to be generated early in a frame to avoid losing opportunity of improving performance. This requires online estimation

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

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
        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 2
            end if
        end if
    else
        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 2
            end if
        end if
    end if
end if
```

ALGORITHM 2: Module to Check FE Bottleneck

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

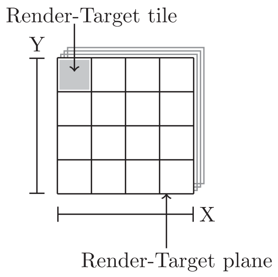

Fig. 7. Render-target plane and tile.

of the projected frame rate of such applications. We present a completely dynamic architecture-independent mechanism for estimating the projected frame rate at any point in time during a frame.

Our frame rate estimation scheme operates in two modes, namely, learning and prediction modes. The learning mode lasts for one full frame. It measures the amount of work in the frame and the rendering time of the frame. In the prediction mode, our scheme starts producing frame rate projections and continuously compares the learning mode data with the data from the newly completed frames. If the newly observed values differ from the learned values by more than a threshold, the hardware discards the learned values and switches back to the learning mode. Unlike prior proposals for estimating GPU progress [20, 64], our proposal does not assume tile-based deferred rendering and does not require any profile information.

4.2.1 Learning Mode. The purpose of using a part of rendering as the learning phase is to quantify the amount of work in a frame and the time needed to render a frame at a given rendering speed. Rendering of a frame involves generating the color values of all pixels into a buffer commonly known as the render target (RT). A single pixel in the RT can get overdrawn multiple times depending on the arrival order and depth of the geometry primitives. 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 rendering of a frame into render target planes (RTP). As shown in Figure 7, each RTP represents a batch of updates that cover all tiles of the RT.

We maintain a 64-entry RTP information table in the GPU. For a frame, each entry of this table records three pieces of information about a distinct RTP: (i) total number of updates to the RTP, (ii) the number of cycles to finish the RTP, and (iii) the number of RTTs in the RTP. Our implementation assumes each of the three fields to be four bytes in size. We use the number of updates to the RTP and the RTT count during the prediction mode of the algorithm to determine the amount of work left in a frame. The number of cycles needed by an RTP is used to obtain the average number of cycles per RTP. This is used to compute the expected frame 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.

4.2.2 Prediction Mode. 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 $F_i = C^i_{rtp} \times N^i_{rtp}$. We obtain N^i_{rtp} directly from the data collected in the learning mode assuming that it doesn't change for the current frame i. To compute C^i_{rtp} for the frame being rendered currently, let the fraction of the frame that has been rendered so far be λ, the average number of cycle per RTP seen in the current frame be C^i_{cur}, and the average number of cycles per RTP recorded in the learning mode be C_{avg}. Therefore, C^i_{rtp} can be computed as $C^i_{rtp} = \lambda \times C^i_{cur} + (1 - \lambda) \times C_{avg}$.

4.3 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 set by our criticality estimation hardware specifying if the access is critical. If the GPU is running a 3D scene rendering workload and the predicted frame rate meets the target, no GPU access is marked critical. Our DRAM scheduling policy uses the criticality information to appropriately share the DRAM access bandwidth between the CPU and the GPU. We propose two DRAM scheduling policies, namely, the GPU-favoring policy and the interference mitigation policy (IM policy). 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 causing the CPU blocks to get replaced at a faster rate 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. Effectively, the CPU prioritization probability in an interval is same as the observed probability that a given CPU access is de-prioritized by a younger critical GPU access in the last 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. In an interval, a CPU request is prioritized over a pending critical GPU request with this probability.

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. H category has more than 70% miss rate, M category has miss rate between 10% and 70%, and L category has miss rate at most 10%. In two consecutive intervals, if an 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 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 depending on the control-flow, cache-friendliness, DRAM bandwidth utilization, and the synchronization primitives used (e.g., intra-block barriers and atomics) during a phase. This is observed particularly in kernels with irregular access patterns. 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 (measured in terms of 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. This mode continues until the GPU performance in the current window becomes equal to the previous window signifying no additional advantage of staying in this mode.

5 SIMULATION ENVIRONMENT

We use a modified version of the Multi2Sim simulator [63] to model the CPU cores of the simulated heterogeneous CMP. Each dynamically scheduled out-of-order issue ×86 core is clocked at 4GHz. We use two GPU simulators, one to execute the 3D rendering jobs and the other to execute the CUDA applications. The 3D rendering GPU is modeled with an upgraded version of the Attila GPU simulator [40]. 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 shader throughput of the GPU is one tera-FLOPS (single precision). The GPU model used for CUDA applications is borrowed from the MacSim infra-structure [30]. Since the CUDA applications make use of the shader cores only, the GPU simulator contains a detailed model of the shader core island. The shader throughput of this GPU is 512 GFLOPS (single precision). Depending on the type of the GPU workload, one of the two GPU models gets attached to the rest of the CMP. The DRAM modules are modeled using DRAMSim2 [53]. Table 1 presents the detailed configuration.

The heterogeneous workloads are prepared by mixing the SPEC CPU 2006 applications with 3D scene rendering jobs drawn from fourteen popular DirectX 9 and OpenGL game titles as well as six CUDA applications. The DirectX and OpenGL API traces are obtained from the Attila simulator distribution and the 3DMark06 suite [67]. The simulated multi-frame game regions are selected after skipping over the initial sequence. We select thirteen SPEC CPU 2006 applications and partition them into two groups 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. The multi-frame 3D rendering jobs are detailed in Table 2. The last column of this table lists the baseline average frames per second (FPS) achieved by the applications when co-scheduled with the four-way CPU mixes. The CUDA applications are shown in Table 3. LBM is drawn from Parboil [58]; CFD and BFS from Rodinia 3.0 [5, 6]; FASTWALSH, BLACKSCHOLES, and REDUCTION from the CUDA SDK 4.2. These six CUDA applications are selected based on their sensitivity toward memory system optimizations.

Within each heterogeneous mix, the first 200M instructions retired by each CPU core are used to warm up the caches. After the warm-up, each CPU application in a mix commits at least 450M dynamic instructions [54]. Early-finishing applications continue to run until each CPU application

Table 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 GPGPU
Shader cores: 16, 2 GHz, sixteen SP FLOPs/cycle
Instruction, data cache per core: 4 KB, 32 KB, Texture, constant cache per core: 8 KB, 8 KB, Software-managed shared memory per core: 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 [19]
Interconnect: bi-directional 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

commits its representative set of dynamic instructions and the GPU completes its job. The performance metrics used for CPU mix, 3D animation, and CUDA application are respectively weighted speedup, average frame rate, and the number of execution cycles.

5.1 Additional Hardware Overhead

In this section, we discuss the hardware overhead of our proposal. 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 has 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 frame rate estimation mechanism maintains a 64-entry RTP information table,

Table 2. Graphics Frame Details

Application, DX/OGL[7]	Frames	Res.[8]	FPS
3DMark06 GT1, DX	670–671	R1	5.9
3DMark06 GT2, DX	500–501	R1	14.0
3DMark06 HDR1, DX	600–601	R1	16.7
3DMark06 HDR2, DX	550–551	R1	21.8
Call of Duty 2 (COD2), DX	208–209	R2	19.5
Crysis, DX	400–401	R2	6.7
DOOM3, OGL	300–314	R3	80.7
Half Life 2 (HL2), DX	25–33	R3	77.4
Left for Dead (L4D), DX	601–605	R1	33.6
Need for Speed (NFS), DX	10–17	R1	66.6
Quake4, OGL	300–309	R3	80.5
Chronicles of Riddick (COR), OGL	253–267	R1	103.9
Unreal Tournament 2004 (UT2004), OGL	200–217	R3	132.5
Unreal Tournament 3 (UT3), DX	955–956	R1	26.6

[7]DX=DirectX, OGL=OpenGL.

[8]Resolutions: R1=1280 × 1024, R2=1920 × 1200, R3=1600 × 1200.

Table 3. 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

each entry being 97 bits leading to an overhead of under 1KB. Overall, the storage overhead of our proposal is only 3.1KB. 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.

6 SIMULATION RESULTS

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. Sections 6.1 and 6.2 respectively discuss the results for the mixes containing the 3D rendering and CUDA workloads.

6.1 Mixes with 3D Rendering Workloads

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

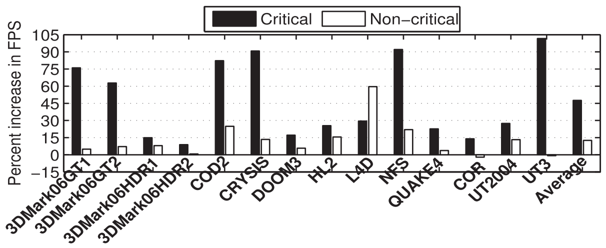

Fig. 8. Percent improvement in FPS when LLC behaves ideally for critical and non-critical accesses.

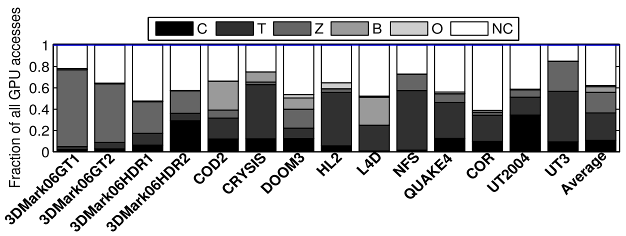

Fig. 9. Distribution of critical accesses.

6.1.1 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 8 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 identified by our algorithm with an ideal memory sub-system (LLC onward) 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. This points to further scope of improvement in understanding blitter criticality. 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 9 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 3 is among the dominant critical streams identified by our algorithm. This confirms the success of our algorithm.

6.1.2 Frame Rate Estimation. Figure 10 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. 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%.

6.1.3 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

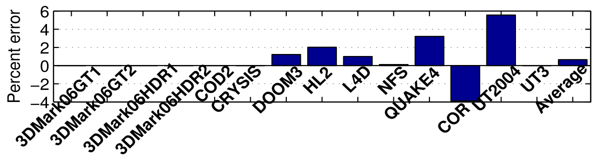

Fig. 10. Percent error in frame rate estimation.

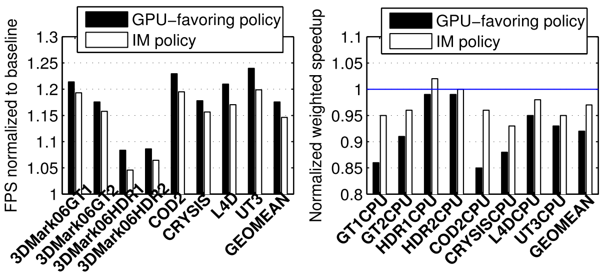

Fig. 11. Left: normalized FPS of GPU applications that perform below target FPS. Right: weighted CPU speedup for the mixes.

set this target to 40 FPS and show the results for the eight applications that deliver frame rate below this level (see Table 2).[9] Figure 11 evaluates the GPU-favoring and IM policies (Section 4.3) 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*. Each bar represents an average (geometric mean) of three heterogeneous mixes that each GPU workload belongs to. 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 IM-LLC and IM-SCHED work together in the IM policy.

To further understand the quality of the critical access set identified by our algorithm, we conduct two experiments with the HL mixes containing the GPU applications with lower than 40 FPS. In the first experiment, we evaluate the FPS improvement when out of the critical accesses, as identified by our algorithm, a randomly selected 25% or 50% or 75% or 100% population is marked critical. The left panel of Figure 12 shows the stacked improvement in FPS as a new quarter of the critical accesses is marked critical. These results show that all quarters are equally important from performance viewpoint. In the second experiment, we explore if our criticality estimation algorithm can be replaced by a simpler random sampling algorithm that marks accesses as critical uniformly at random while maintaining the total number of critical accesses from each stream

[9]Our prior work has explored mechanisms to improve the system performance of the heterogeneous mixes containing the remaining GPU applications that already meet the 40 FPS target. The central idea involved throttling the LLC access rate of the GPU application so that it delivers just 40 FPS and shifting the memory system resources (LLC capacity and DRAM bandwidth) thus freed to the co-executing CPU applications so that they improve in performance [50].

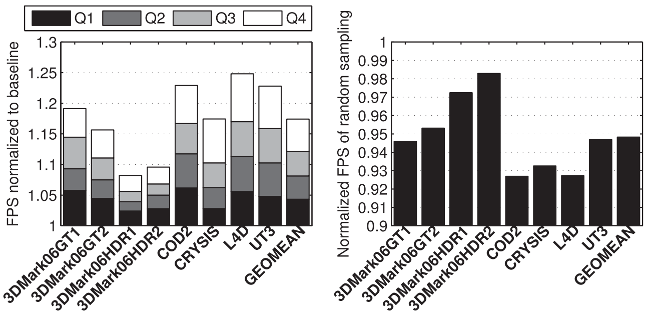

Fig. 12. Left: cumulative performance contribution of each quarter of the critical accesses. Right: performance of random sampling normalized to the proposed criticality estimation algorithm.

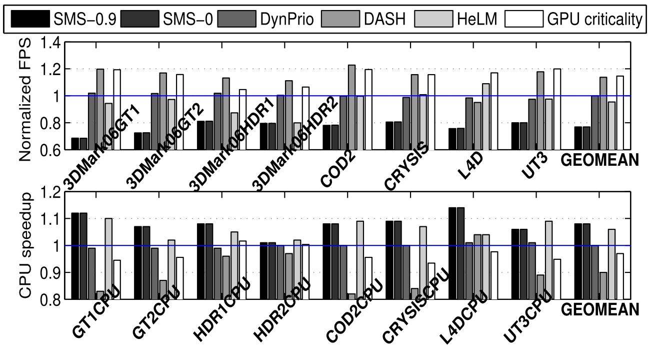

Fig. 13. Top: FPS speedup over baseline. Bottom: weighted CPU speedup for the mixes.

same as our algorithm. The right panel of Figure 12 shows the performance of this algorithm normalized to our algorithm. On average, the random sampling technique performs 5% worse than our algorithm.

6.1.4 Comparison to Related Proposals. Several DRAM scheduling policies have been proposed for heterogeneous CMPs. These proposals were discussed in Section 2. We compare our proposal against staged memory scheduling (SMS) [2], dynamic priority scheduler (DynPrio) [20], and deadline-aware scheduling (DASH) [64]. 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. SMS-0.9 is expected to favor latency-sensitive jobs while SMS-0 is expected to favor bandwidth-sensitive jobs. We let DynPrio and DASH use 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 [39]. This policy employs LLC bypassing for a subset of GPU read misses to create LLC space for CPU as well as GPU.

Figure 13 shows the comparison for the heterogeneous mixes containing the GPU applications that fail to meet the target FPS. SMS suffers 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

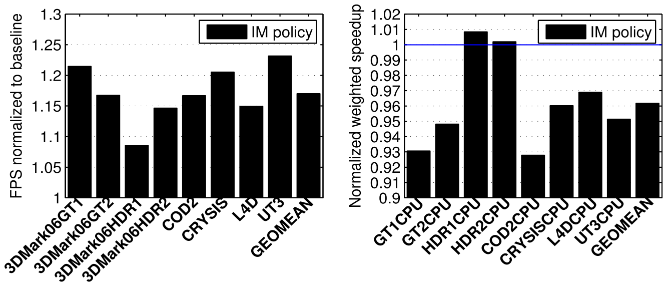

Fig. 14. Left: normalized FPS of GPU applications that perform below target FPS. Right: weighted CPU speedup for the mixes.

co-scheduled CPU mixes by 10% on average (lower panel of Figure 13). 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. Referring back to Figure 11, we notice that our GPU-favoring policy, which always prioritizes the critical GPU accesses, performs better than DASH for both GPU and CPU workloads. These results clearly bring out the advantage of prioritizing only the critical GPU accesses as opposed to prioritizing all GPU accesses, as DASH does. Both SMS-0.9 and SMS-0 improve CPU mix performance by 8%, while suffering large losses in GPU performance. HeLM improves CPU performance by 6% on average, while degrading GPU performance by 5%. These results clearly indicate that the GPU performance can be traded off to improve CPU performance and vice-versa in such heterogeneous platforms. Our proposal strikes a nice balance in this trade-off by probabilistically offering express DRAM bandwidth to a subset of the critical GPU accesses while shifting the remaining DRAM bandwidth to the CPU. 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 normalized weighted speedup of the CPU mix [36]. 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%.

6.1.5 *Sensitivity to LLC Capacity.* Figure 14 summarizes the performance of the IM policy when the heterogeneous CMP is equipped with an 8MB shared LLC (as opposed to 16MB 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 11, we observe that for a 16MB LLC, the GPU gain is 15% and the CPU mixes perform within 3% of the baseline, on average.

6.2 Mixes with GPGPU Workloads

Figure 15 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 16MB shared LLC.[10] The left panel quantifies the speedup experienced by the GPU application in the mix, while the right panel shows the weighted speedup of the co-running CPU mixes normalized to the baseline. Both SMS-0.9 and SMS-0 degrade GPU performance (left panel) by 4% on average

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

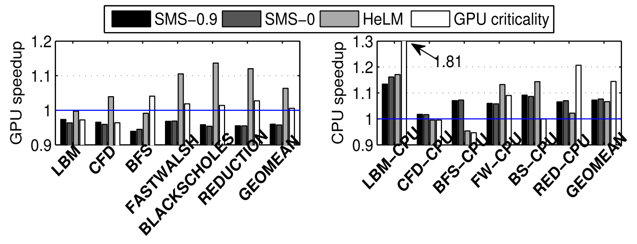

Fig. 15. Left: GPU application speedup. Right: weighted CPU speedup for the mixes.

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.

7 SUMMARY

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 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.

REFERENCES

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

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

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

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

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

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

[7] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi. 2013. 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*. 107–118.

[8] M. Demler. 2013. Iris Pro Takes On Discrete GPUs. In *Microprocessor Report*.

[9] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. 2010. 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*. 353–364.

[10] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. 2010. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*. 335–346.

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

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

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

[14] 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. 2014. Haswell: The Fourth Generation Intel Core Processor. In *IEEE Micro*, 34, 2, 6–20.

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

[16] I. Hur and C. Lin. 2016. Adaptive History-Based Memory Schedulers. In *Proceedings of the 37th International Symposium on Microarchitecture*. 343–354.

[17] 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.

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

[19] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. 2010. High Performance Cache Replacement using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th International Symposium on Computer Architecture*. 60–71.

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

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

[22] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. 2013. 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*. 395–406.

[23] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. 2016. Exploiting Core Criticality for Enhanced GPU Performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Science (SIGMETRICS)*. 351–363.

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

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

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

[27] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das. 2014. Managing GPU Concurrency in Heterogeneous Architectures. In *Proceedings of the 47th International Symposium on Microarchitecture*. 114–126.

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

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

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

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

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

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

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

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

[36] J. Lee and H. Kim. 2012. 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*. 91–102.

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

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

[39] V. Mekkat, A. Holey, P.-C. Yew, and A. Zhai. 2013. Managing Shared Last-level Cache in a Heterogeneous Multicore Processor. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 225–234.

[40] V. Moya, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa. 2006. 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*. 231–241. Source and traces available at http://attila.ac.upc.edu/wiki/index.php/Main_Page.

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

[42] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. 2003. 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*. 129–140.

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

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

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

[46] 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*. 208–222.

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

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

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

[50] S. Rai and M. Chaudhuri. 2017. Improving CPU Performance through Dynamic GPU Access Throttling in CPU-GPU Heterogeneous Processors. In *Proceedings of the 26th IEEE International Heterogeneity in Computing Workshop*. 18–29.

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

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

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

[54] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. 45–57.

[55] 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.

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

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

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

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

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

[61] L. Subramanian, V. Seshadri, A. Ghosh, S. M. Khan, and O. Mutlu. 2015. 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*. 62–75.

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

[63] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. 2012. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the 21st International Conference on Parallel Architecture and Compilation Techniques*. 335–344.

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

[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] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. 2011. A Fully Integrated Multi-CPU, GPU, and Memory Controller 32 nm Processor. In *Proceedings of the International Solid-State Circuits Conference*. 264–266.

[67] 3D Mark Benchmark. http://www.3dmark.com/.

Received April 2017; revised June 2017; accepted June 2017