Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems

Rachata Ausavarungnirun[†] Kevin Kai-Wei Chang[†] Lavanya Subramanian[†] Gabriel H. Loh[‡] Onur Mutlu[†]

[†]Carnegie Mellon University
{rachata,kevincha,lsubrama,onur}@cmu.edu

[‡]Advanced Micro Devices, Inc.
gabe.loh@amd.com

Abstract

When multiple processor (CPU) cores and a GPU integrated together on the same chip share the off-chip main memory, requests from the GPU can heavily interfere with requests from the CPU cores, leading to low system performance and starvation of CPU cores. Unfortunately, state-of-the-art application-aware memory scheduling algorithms are ineffective at solving this problem at low complexity due to the large amount of GPU traffic. A large and costly request buffer is needed to provide these algorithms with enough visibility across the global request stream, requiring relatively complex hardware implementations.

This paper proposes a fundamentally new approach that decouples the memory controller's three primary tasks into three significantly simpler structures that together improve system performance and fairness, especially in integrated CPU-GPU systems. Our three-stage memory controller first groups requests based on row-buffer locality. This grouping allows the second stage to focus only on inter-application request scheduling. These two stages enforce high-level policies regarding performance and fairness, and therefore the last stage consists of simple per-bank FIFO queues (no further command reordering within each bank) and straightforward logic that deals only with low-level DRAM commands and timing.

We evaluate the design trade-offs involved in our Staged Memory Scheduler (SMS) and compare it against three state-of-the-art memory controller designs. Our evaluations show that SMS improves CPU performance without degrading GPU frame rate beyond a generally acceptable level, while being significantly less complex to implement than previous application-aware schedulers. Furthermore, SMS can be configured by the system software to prioritize the CPU or the GPU at varying levels to address different performance needs.

1 Introduction

With increasing number of cores in modern chip multiprocessor (CMP) systems, the main memory system has become a critical shared resource. Memory requests from multiple cores interfere with each other, and this inter-application interference is a significant impediment to individual application and overall system performance. Previous work on application-aware memory scheduling [14, 15, 24, 25] has addressed the problem by making the memory controller (MC) aware of application characteristics and appropriately prioritizing memory requests to improve system performance and fairness.

Recent systems [4, 10, 27] present an additional challenge by introducing integrated graphics processing units (GPUs) on the same die with CPU cores. GPU applications typically demand significantly more memory bandwidth than CPU applications due to a GPU's ability to execute a large number of parallel threads. A GPU uses single-instruction multiple-data (SIMD) pipelines to concurrently execute multiple threads, where a group of threads executing the same instruction is called a *wavefront* or a *warp*. When a wavefront stalls on a memory instruction, a GPU core hides this memory access latency by switching to another wavefront to avoid stalling the pipeline. Therefore, there can be thousands of outstanding memory requests from across all of the wavefronts. This is fundamentally more memory intensive than CPU memory traffic, where each CPU application has a much smaller number of outstanding requests, due to the sequential execution model and limited instruction window size of CPU cores.

Recent memory scheduling research has focused on memory interference between applications in CPU-only scenarios. These past proposals are built around a single centralized request buffer at each MC.[1] The scheduling algorithm implemented in the MC analyzes the stream of requests in the centralized request buffer to determine application memory characteristics, decides on a priority for each core, and then enforces these priorities. Observable memory characteristics may include the number of requests that result in row-buffer hits [15], the bank-level parallelism of each core [15, 25], memory request rates [14, 15], overall fairness metrics [24, 25], and other information. Figure 1(a) shows the CPU-only scenario where the request buffer holds requests only from the CPU cores. In this case, the MC sees a number of requests from the CPU cores and has visibility into their memory behavior. On the other hand, when the request buffer is shared between the CPU cores and the GPU, as shown in Figure 1(b), the large volume of requests from the GPU occupies a significant fraction of the MC's request buffer, thereby limiting the MC's visibility of the CPU applications' differing memory behavior.

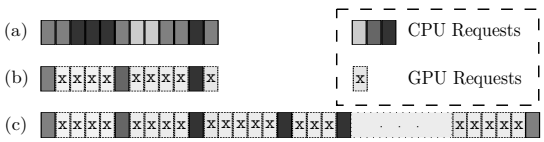

Figure 1. Limited visibility example due to small buffer size. (a) CPU-only case, (b) MC's visibility when CPU and GPU are combined, (c) Improved visibility with a larger request buffer

One approach to increasing the MC's visibility is to increase the size of its request buffer. This allows the MC to observe more requests from the CPU cores to better characterize their memory behavior, as shown in Figure 1(c). For instance, with a large request buffer, the MC can identify and service multiple requests from one CPU core to the same row such that they become row-buffer hits, however, with a small request buffer as shown in Figure 1(b), the MC may not even see these requests at the same time because the GPU's requests have occupied the majority of the request buffer entries.

Unfortunately, very large request buffers impose significant implementation challenges: analyzing many requests, and assigning and enforcing priorities require large structures with high circuit complexity that consume significant die area and power. Therefore, while building a very large, centralized MC request buffer could lead to good application-aware memory scheduling decisions, the resulting area, power, timing and complexity costs are unattractive.

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

[1]This buffer can be partly partitioned across banks, but each bank buffer is centralized for all applications.

three stages of our proposed MC architecture, are: 1) detection of basic within-application memory characteristics (e.g., row-buffer locality), 2) prioritization across applications running on both CPU cores and GPU, and enforcement of policies to reflect the application priorities, and 3) low-level command scheduling (e.g., activate, precharge, read/write), enforcement of device timing constraints (e.g., t_{RAS}, t_{FAW}, etc.), and resolution of resource conflicts (e.g., data bus arbitration).

Our specific SMS implementation makes widespread use of distributed FIFO structures to maintain a simple implementation. SMS can at the same time provide fast service to low memory-intensity (likely latency-sensitive) applications and effectively exploit row-buffer locality and bank-level parallelism for high memory-intensity (bandwidth-demanding) applications. While SMS provides a specific implementation of a multi-stage memory controller, our staged approach for MC organization provides a general framework for exploring scalable memory scheduling algorithms capable of handling the diverse memory needs of integrated CPU-GPU systems and other heterogeneous systems of the future.

This work makes the following contributions:

- We identify and present the challenges posed to existing memory scheduling algorithms due to the high memory-bandwidth demands of GPU applications.
- We propose a new decentralized, *multi-stage* approach to application-aware memory scheduling that effectively handles the interference caused by bandwidth-intensive applications to other applications, while simplifying the hardware implementation by decoupling different functional tasks of a memory controller across multiple stages.
- We evaluate our proposed approach against three previous memory scheduling algorithms [14, 15, 31] across a wide variety of workloads and CPU-GPU systems and show that it improves CPU performance without degrading GPU frame rate beyond an acceptable level, while providing a design that is significantly less complex to implement. Furthermore, SMS can be dynamically configured by the system software to prioritize the CPU or the GPU at varying levels to address different performance needs.

2 Background

In this section, we provide a review of DRAM organization and discuss how past research dealt with the challenges of providing performance and fairness for modern memory systems.

2.1 Main Memory Organization

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

Each two-dimensional array of DRAM cells constitutes a bank. A DRAM chip consists of multiple banks. Multiple DRAM chips are put together and operated in lockstep to form a rank. One or more ranks form a channel. All banks on a channel share a common set of command and data buses, and the MC that controls that memory channel is responsible for scheduling commands such that each bus is used by only one bank at a time. Operations on multiple banks may occur in parallel (e.g., opening a row in one bank while reading data from another bank's row buffer) so long as the commands are properly scheduled and any other DRAM timing constraints are obeyed. An MC can improve memory system throughput by scheduling requests such that bank-level parallelism, or BLP (i.e., the number of banks simultaneously busy responding to commands), is increased. A key challenge in the implementation of modern, high-performance MCs is to effectively improve system performance by maximizing both row-buffer hits and BLP while simultaneously providing fairness among multiple CPU cores and the GPU.

2.2 Memory Request Scheduling

Accessing off-chip memory is a major performance bottleneck in microprocessors. The MC is responsible for buffering and servicing memory requests from the different cores and the GPU. Typical implementations make use of a memory request buffer to hold and keep track of all in-flight requests. Scheduling logic then decides which requests should be serviced and issues the corresponding commands to the DRAM devices.

As all of the cores must share the limited off-chip memory bandwidth, a large number of outstanding requests greatly increases contention for the memory data and command buses. Because a bank can only process one command at a time, a large number of requests also increases bank contention, where requests must wait for busy banks to finish servicing other requests. A request from one core can also cause a row buffer containing data for another core to be closed, thereby reducing the row-buffer hit rate of that other core (and vice-versa). All of these effects increase the latency of memory requests by both increasing queuing delays (time spent waiting for the MC to start servicing a request) and DRAM device access delays (due to decreased row-buffer hit rates and bus contention). As a result, different memory scheduling algorithms have been designed to service memory requests in an order different from the order in which the requests arrived at the MC, to increase row-buffer hit rates, bank level parallelism, fairness, or to achieve other goals.

2.3 Memory Request Scheduling in CPU-only Systems

Memory scheduling algorithms can improve system performance by reordering memory requests to deal with the different constraints of DRAM. The first-ready first-come-first-serve (FR-FCFS) [31, 37] algorithm prioritizes requests that result in row-buffer hits, and otherwise prioritizes older requests. FR-FCFS increases DRAM throughput, but it can cause fairness problems by unfairly deprioritizing applications with low row-buffer locality and low memory-intensity, as shown in previous work [21, 24]. Several application-aware memory scheduling algorithms [14, 15, 23, 24, 25] have been proposed to balance both performance and fairness. Parallelism-aware Batch Scheduling (PAR-BS) [25] batches requests based on their arrival times and prioritizes the oldest batch over others, to provide fairness. In a batch, applications are ranked to preserve bank-level parallelism (BLP) within an application's requests. ATLAS [14] prioritizes applications that have received the least memory service. As a result, applications with low memory-intensity, which typically attain low memory service, are prioritized, improving system throughput. However, applications with high memory-intensity are deprioritized and hence slowed down significantly, resulting in unfairness [14]. Thread Cluster Memory scheduling (TCM) [15] addresses this unfairness problem by dynamically clustering applications into low and high memory-intensity clusters based on their memory intensities. To improve system throughput, TCM always prioritizes applications in the low memory-intensity cluster. To prevent unfairness, TCM periodically shuffles priorities of high memory-

intensity applications. TCM was shown to achieve high system performance and fairness in multi-core systems.

2.4 Analysis of GPU Memory Accesses

A typical CPU application has a relatively small number of outstanding memory requests at any time. The size of a processor's instruction window bounds the number of misses that can be simultaneously exposed to the memory system. Branch prediction accuracy limits how large the instruction window size can be usefully increased. In contrast, GPU applications have very different access characteristics. They generate significantly more memory requests than CPU applications. A GPU application can consist of many thousands of parallel threads, where memory stalls in one group of threads can be hidden by switching execution to one of the many other groups of threads. GPUs also use fixed-function units (e.g., texture units, z (depth) units, and color units) that are responsible for a large number of memory requests.

Figure 2 (a) shows the memory request rates for a representative subset of our GPU applications and SPEC2006 (CPU) applications, as measured by memory requests per thousand instructions when each application runs alone on the system.[2] The memory access intensity of the GPU applications is often multiple times higher than that of the SPEC benchmarks. The most intensive CPU application we study (mcf) has 42.8% of the memory intensity of the least memory-intensive GPU application we study (Bench01). Figure 2 (b) shows the row-buffer hit rates (also called row-buffer locality or RBL). The GPU applications show consistently high levels of RBL, whereas the SPEC benchmarks exhibit more variability. The GPU applications have high levels of spatial locality, often due to access patterns related to large sequential memory accesses (e.g., frame buffer updates). Figure 2(c) shows the average BLP for each application, demonstrating that the GPU applications access many banks at the same time.

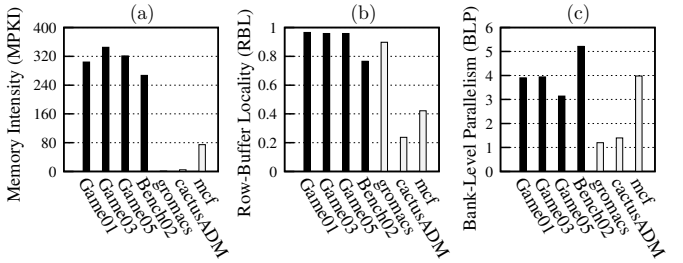

Figure 2. Memory access characteristics. (a) Memory intensity, measured by memory requests per thousand instructions, (b) row-buffer locality, measured by the fraction of accesses that hit in the row buffer, and (c) bank-level parallelism, measured by average number of banks kept busy when there is at least one request outstanding.

In addition to the high-intensity memory traffic of GPU applications, there are other properties that distinguish GPU applications from CPU applications. Kim et al. [15] observed that CPU applications with streaming access patterns typically exhibit high RBL but low BLP, while applications with less uniform access patterns typically have low RBL but high BLP. In contrast, GPU applications have *both* high RBL and high BLP. The combination of high memory intensity, high RBL and high BLP means that the GPU will cause significant interference to other applications across all banks, especially when using a memory scheduling algorithm that preferentially favors requests that result in row-buffer hits.

2.5 Memory Request Scheduling in GPU-only Systems

As opposed to CPU applications, GPU applications are not very latency sensitive, as there are a large number of independent threads to tolerate long memory latencies. However, the GPU requires a significant amount of memory bandwidth far exceeding even the most memory-intensive CPU applications, as we have shown in Figure 2. Previous works [2, 3, 35] have proposed memory request scheduling policies for GPUs and GPGPUs. However, to our knowledge, no previous work has explored application-aware memory scheduling in the context of *integrated CPU-GPU systems*. In particular, as described in Section 2.4, GPU memory accesses can cause significant interference to other CPU applications since GPU applications have the combination of high memory intensity, high RBL, and high BLP. Our goal in this work is to design a new application-aware scheduler that mitigates inter-application interference to improve system performance and fairness in heterogeneous CPU-GPU systems.

[2]Section 5 describes our simulation methodology.

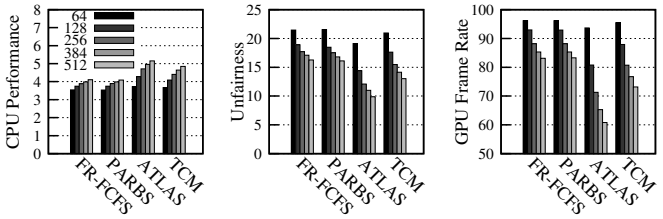

Figure 3. Performance and unfairness of different memory scheduling algorithms with varying request buffer sizes, averaged over 35 workloads.

3 Challenges with Existing Memory Controllers

3.1 Shortcomings of Previous Mechanisms

As discussed in Section 2, previously proposed application-aware memory scheduling algorithms (for CPU-only systems) are designed to improve system performance and fairness by prioritizing some applications over others, based on applications' memory access characteristics. However, when faced with CPU and GPU applications that have vastly disparate memory access characteristics, these scheduling algorithms are non-robust, due to specific aspects in each of their designs, and perform poorly. Figure 3 shows CPU performance, unfairness, and GPU frame rate of different previously proposed memory schedulers with varying request buffer sizes for a 16-core CPU/1-GPU system. These data are averaged over 35 workloads; details of our methodology are in Section 5. FR-FCFS [31] provides high GPU frame rate. However, it provides poor CPU performance and fairness, as it always prioritizes the row-buffer hits of the GPU. ATLAS [14] provides high CPU performance and fairness compared to other scheduling mechanisms; however, it significantly degrades the frame rate of the GPU. As CPU applications have lower memory intensities than their GPU counterparts, ATLAS prioritizes CPU applications over GPU applications, which leads to lower GPU frame rate. TCM [15] provides good CPU performance and GPU frame rate; however, it is highly unfair because of the way it clusters applications. TCM places some high-memory-intensity CPU applications in the low memory-intensity cluster, unfairly prioritizing them over other high-memory-intensity applications that are placed in the high-memory-intensity cluster. In the light of these shortcomings of previous memory scheduling algorithms, we aim to design a fundamentally new memory scheduler that can robustly provide good system performance (*both CPU performance and GPU performance together*) as well as high fairness to CPU applications.

3.2 The Need for Request Buffer Capacity

The non-robustness of previously proposed CPU-only scheduling algorithms is further magnified when there are a small number of entries in the request buffer, as the memory-intensive GPU application hogs a large number of request buffer entries. The results from Figure 2 show that a typical GPU application has a very high memory intensity. As discussed in Section 1, the large number of GPU memory requests occupy many of the MC's request buffer entries, thereby making it difficult for the

MC to properly determine the memory access characteristics of each of the CPU applications. Figure 3 shows the performance impact of increasing the MC's request buffer size for a variety of memory scheduling algorithms. As can be observed, with increased visibility within larger request buffers, CPU performance improves (while GPU performance does not degrade beyond a generally acceptable level).

3.3 Implementation Challenges in Providing Request Buffer Capacity

Figure 3 shows that when the MC has enough visibility across the global memory request stream to properly characterize the behavior of each core, a sophisticated application-aware algorithm like TCM improves CPU performance and fairness by making better scheduling decisions compared to FR-FCFS, which is application-unaware. Unfortunately, implementing a sophisticated algorithm like TCM over such a large request buffer introduces significant implementation challenges. For all algorithms that use a centralized request buffer and prioritize requests that result in row-buffer hits (FR-FCFS, PAR-BS, ATLAS, TCM), content associative memories (CAMs) are needed for each request buffer entry, to compare its requested row against the currently open row in the corresponding DRAM bank. For all algorithms that prioritize requests based on rank/age (FR-FCFS, PAR-BS, ATLAS, TCM), a large comparison tree (priority encoder) is needed to select the highest ranked/oldest request from all request buffer entries. The size of this comparison tree grows with request buffer size. Furthermore, in addition to this logic for reordering requests and enforcing ranking/age, TCM also requires additional logic to continually monitor each core's last-level cache MPKI (note that a CPU core's instruction count is not typically available at the MC), each core's RBL, which requires additional *shadow row buffer index* tracking [6, 24], and each core's BLP.

Apart from the logic required to implement the policies of the specific memory scheduling algorithms, all of these MC designs need additional logic to enforce DRAM timing constraints. Note that different timing constraints will apply depending on the *state* of each memory request. For example, if a memory request's target bank currently has a different row loaded in its row buffer, then the MC must ensure that a precharge (row close) command is allowed to issue to that bank (e.g., has t_{RAS} elapsed since the row was opened?), but if the row is already closed, then different timing constraints will apply. For each request buffer entry, the MC will determine whether or not the request can issue a command to the DRAM based on the current state of the request and the current state of the DRAM system. The DRAM timing-constraint checking logic (including data and command bus availability tracking) needs to keep track of the commands that come from *every* request buffer entry. This type of monolithic MC effectively implements a large out-of-order scheduler. Note that typical instruction schedulers in modern out-of-order processors have only 32 to 64 entries [9]. Even after accounting for the clock speed differences between CPU core and DRAM command frequencies, it is very difficult to implement a fully-associative,[3] age-ordered/prioritized, out-of-order scheduler with hundreds of entries [28]; the resulting complexity, cycle time, and power consumption would be unnecessarily or prohibitively high.

Our goal in this paper is to devise an easier-to-implement, scalable, application-aware memory scheduling algorithm that provides high system performance and fairness in heterogeneous CPU-GPU systems. To this end, we devise both 1) a new scheduling algorithm and 2) new principles with which a memory scheduler can be organized and simply implemented.

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

4 The Staged Memory Scheduler

Overview: Our proposed SMS architecture introduces a new memory controller (MC) design that provides 1) scalability and simpler implementation by decoupling the primary functions of an application-aware MC into a simpler multi-stage MC, and 2) performance and fairness improvement by reducing the interference from bandwidth-intensive applications. Specifically, SMS provides these benefits by introducing a three-stage design. The first stage is the *batch formation* stage that groups requests from the same application that access the same row to improve row-buffer locality. The second stage is the *batch scheduler* that schedules batches of requests across different applications. The last stage is the *DRAM command scheduler* that schedules requests while satisfying all DRAM constraints.

The staged organization of SMS lends directly to a low-complexity hardware implementation. Figure 4 illustrates the overall hardware organization of the SMS.

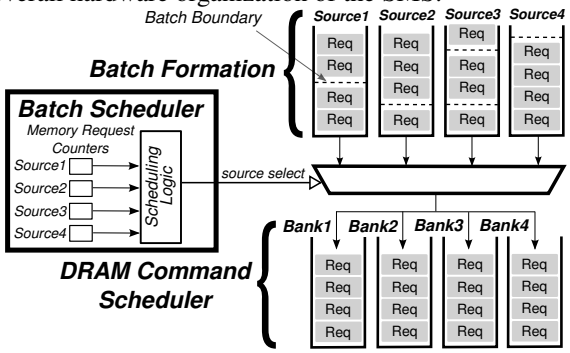

Figure 4. Organization of the SMS.

In Sections 4.1 and 4.2, we describe the algorithm used in each stage. Section 4.3 describes the rationale behind these algorithms. Section 4.4 describes the hardware implementation of SMS. Section 4.5 qualitatively compares SMS to previously proposed scheduling algorithms.

4.1 The SMS Algorithm

Stage 1 - Batch Formation: The goal of this stage is to combine individual memory requests from each source into batches of row-buffer hitting requests. It consists of several simple FIFO structures, one per *source* (i.e., a CPU core or the GPU). Each request from a given source is initially inserted into its respective FIFO upon arrival at the MC. A *batch* is simply one or more memory requests from the same source that access the same DRAM row. That is, all requests within a batch, except perhaps for the first one, would be row-buffer hits if scheduled consecutively. A batch is deemed complete or *ready* when an incoming request accesses a different row, when the oldest request in the batch has exceeded a threshold age, or when the FIFO is full. Only ready batches are considered by the second stage of SMS.
Stage 2 - Batch Scheduler: The batch scheduler deals directly with batches, and therefore need not worry about scheduling to optimize for row-buffer locality. Instead, the batch scheduler can focus on higher-level policies regarding inter-application interference and fairness. The goal of the batch scheduler is to prioritize batches from applications that are latency critical, while making sure that bandwidth-intensive applications (e.g., the GPU) still make reasonable progress.

The batch scheduler operates in two states: *pick* (a batch to send downstream) and *drain* (the picked batch of requests). In the pick state, the batch scheduler considers every source FIFO (from stage 1) that contains a ready batch. It picks one ready batch based on either a shortest job first (SJF) or a round-robin policy. Using the SJF policy, the batch scheduler chooses the oldest ready batch from the source with the fewest total in-flight memory requests across all three stages of the SMS. SJF prioritization reduces average request service latency, and it tends to favor latency-sensitive applications, which tend to have fewer total

requests. Using the round-robin policy, the batch scheduler simply picks the next ready batch in a round-robin manner across the source FIFOs. This ensures that high memory-intensity applications receive adequate service. Overall, the batch scheduler uses the SJF policy with a probability p and the round-robin policy otherwise. p is a configurable parameter, which is explained in detail in Section 4.2.

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

Stage 3 - DRAM Command Scheduler (DCS): The DCS consists of one FIFO queue per DRAM bank (e.g., eight banks/FIFOs per rank for DDR3). The drain state of the batch scheduler places the memory requests directly into these FIFOs. Note that because batches are moved into the DCS FIFOs one batch at a time, row-buffer locality within a batch is preserved within a DCS FIFO. At this point, any higher-level policy decisions have already been made by the batch scheduler; therefore, the DCS simply issues low-level DRAM commands, ensuring DRAM protocol compliance.

In any given cycle, the DCS considers only the requests at the *head* of each of the per-bank FIFOs. For each request, the DCS determines whether that request can issue a command based on the request's current row-buffer state (e.g., is the row buffer already open with the requested row?) and the current DRAM state (e.g., time elapsed since a row was opened in a bank, and data bus availability). If more than one request is eligible to issue a command in any given cycle, the DCS arbitrates across each DRAM bank in a round-robin fashion.

4.2 Additional Algorithm Details

Batch Formation Thresholds: The batch formation stage holds requests in the per-source FIFOs until a complete batch is ready. This could unnecessarily delay requests, as the batch will not be marked ready until a request to a *different* row arrives at the MC, or the FIFO becomes full. This additional queuing delay can be particularly detrimental to the performance of low memory-intensity, latency-sensitive applications.

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

For medium memory-intensity (1-10 MPKC) and high memory-intensity ($>$10 MPKC) applications, the batch formation stage uses age thresholds of 50 and 200 cycles, respectively. That is, regardless of how many requests are in the current batch, when the oldest request's age exceeds the threshold, the entire batch is marked ready (and consequently, any new requests that arrive, even if accessing the same row, will be grouped into a new batch).[4]

Request Bypass: We observe that there are two instances when it is beneficial to let requests bypass the batch formation and scheduling stages and proceed directly to the DCS per-bank FIFOs: 1) the latency-sensitive requests of applications with low memory-intensity, as described in Section 4.1 and 2) all applications' requests when the system is lightly loaded, such that requests are not unnecessarily delayed in the batch formation FIFOs, while banks are potentially idle. This latter bypass is enabled whenever the total number of in-flight requests (across *all* sources) in the DCS is less than sixteen.

SJF Probability: As described above, the batch scheduler uses the SJF policy with probability p and the round-robin policy with probability $1-p$. The value of p determines whether the CPU or the GPU receives higher priority. When p is high, the SJF policy is applied more often and applications with fewer outstanding requests are prioritized. Hence, the batches of the likely less memory-intensive CPU applications are prioritized over the batches of the GPU application. On the other hand, when p is low, request batches are scheduled in a round-robin fashion more often. Hence, the memory-intensive GPU application's request batches are likely scheduled more frequently, and the GPU is prioritized over the CPU. Different systems (and maybe even the same system at different times) could have different performance needs. In some systems, CPU performance could matter more than GPU performance, while in other systems, GPU performance could matter more. Therefore, we make p a dynamically configurable parameter that can be tuned to appropriately prioritize the CPU or the GPU based on the system's needs. We evaluate sensitivity to p and its configurability in Section 6.2.1.

4.3 SMS Rationale

In-Order Batch Formation: It is important to note that batch formation occurs in the order of request arrival. This potentially sacrifices some row-buffer locality, as requests to the same row may be interleaved with requests to other rows. We considered many variations of batch formation that allowed out-of-order grouping of requests, to maximize the length of a run of row-buffer hitting requests, but the overall performance benefit was not significant.[5] First, constructing very large batches of row-buffer hitting requests can introduce significant unfairness as other requests may need to wait a long time for a bank to complete its processing of a long run of row-buffer hitting requests [21, 24]. Second, row-buffer locality across batches may still be exploited by the DCS. For example, consider a core that has three batches accessing row X, row Y, and then row X again. If X and Y map to different DRAM banks, say banks A and B, then the batch scheduler will send the first and third batches (row X) to bank A, and the second batch (row Y) to bank B. Within the DCS's FIFO for bank A, the requests for the first and third batches could still all be one after the other, thereby exposing the row-buffer locality across the batches.

In-Order Batch Scheduling from a Per-Source FIFO: Due to contention and back-pressure in the system, it is possible that a FIFO in the batch formation stage contains more than one valid batch. In such a case, it could be desirable for the batch scheduler to pick one of the batches not currently at the head of the FIFO. For example, the bank corresponding to the head batch may be busy while the bank for another batch is idle. Scheduling batches out of order could decrease the service latency for the later batches, but in practice we found that it does not make a big difference[6] and adds significant implementation complexity. It is important to note that even though batches are dequeued from the batch formation stage in arrival order per FIFO, request ordering can still slip *between* FIFOs. For example, the batch scheduler may choose a recently arrived (and formed) batch from a high-priority (i.e., latency-sensitive) source even though an older, larger batch from a different source is ready.

In-Order DRAM Command Scheduling: For each of the

[4] Note that while TCM uses the MPKI metric to classify memory intensity, SMS uses misses per thousand *cycles* (MPKC) since the per-application instruction counts are not typically available in the MC. While it would not be overly difficult to expose this information, this is just one more implementation overhead that SMS can avoid. We also evaluated our mechanism using MPKI, and we found that it provides similar results.

[5] We evaluated our mechanism with out-of-order batch formation. Out-of-order provides $<$5% performance difference over in-order batch formation.

[6] We found there is less than 1% performance difference because per-source FIFOs already keep batches in age order.

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

Table 1. Hardware storage required for SMS.

per-bank FIFOs in the DCS, the requests are already grouped by row-buffer locality (because the batch scheduler drains an entire batch at a time), and arranged in an order that reflects per-source priorities. Further reordering at the DCS could undo the prioritization decisions made by the batch scheduler. The in-order nature of each of the DCS per-bank FIFOs does not prevent out-of-order scheduling at the global level. A CPU's requests may be scheduled to the DCS in arrival order, but the requests may get scattered across different banks, and the issue order among banks may be out-of-order relative to each other.

4.4 Hardware Implementation

Batch Formation: The batch formation stage consists of one FIFO per source. Each FIFO maintains an extra register that records the row index of the last request inserted into the FIFO, so that any incoming request's row index can be compared to determine if the request can be added to the existing batch. Note that this requires only a single comparator (used only once at insertion) per FIFO. Contrast this to a conventional monolithic request buffer where comparisons on *every* request buffer entry are made every cycle, potentially against all currently open rows across all banks.

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

DRAM Command Scheduler: The DCS stage consists of the per-bank FIFOs. The logic to track and enforce the various DRAM timing and power constraints is identical to the case of the monolithic scheduler, but the scale is drastically different. The DCS' DRAM command-processing logic considers only the requests at the head of each of the per-bank FIFOs (eight per rank for DDR3), whereas the monolithic scheduler requires logic to consider *every* request buffer entry.

Request Bypass: Bypass for a low memory-intensity application is enabled by setting its threshold age for batching to zero. This allows incoming requests of the application to fall through the first stage into the batch scheduler. The batch scheduler, by design, prioritizes low memory-intensity applications using the SJF policy. Therefore, no additional logic/wiring is required to bypass low memory-intensity applications' requests. Bypassing requests of all applications when the system is lightly loaded can be accomplished by simply setting both SJF probability and threshold age to zero. This will allow requests from every application to proceed directly into the DCS FIFOs in a round-robin fashion at a rate of one request per cycle. However, request bypass in both of these cases incurs a minimum two-cycle delay as a request needs to spend at least one cycle in the batcher and at least one cycle in the DCS.

Overall Configuration and Hardware Cost: The final configuration of SMS that we use in this paper consists of the following hardware structures whose sizes are empirically determined. The batch formation stage uses 10-entry FIFOs for each of the CPU cores, and a 20-entry FIFO for the GPU. The DCS uses a 15-entry FIFO for each of the eight DDR3 banks. For sixteen CPU cores and a GPU, the aggregate capacity of all of these FIFOs is 300 requests, although at any point in time, the SMS logic can only consider or act on a small subset of the entries (i.e., the seventeen batches at the heads of the batch formation FIFOs and the requests at the heads of the per-bank DCS FIFOs). In addition to these primary structures, there are a small number of bookkeeping counters. One counter per source is needed to track the number of in-flight requests. Counters are also needed to track per-source MPKC rates for memory-intensity classification, which are periodically reset to adapt to phase changes.[7] Table 1 summarizes the hardware overhead required for each stage of SMS.

4.5 Qualitative Comparison with Previous Scheduling Algorithms

We described recently proposed application-aware memory schedulers (PAR-BS [25], ATLAS [14], and TCM [15]) in Section 2. We now qualitatively compare SMS with these schedulers.

PAR-BS batches memory requests based on their ages and cores, and prioritizes all requests from one batch over others. As a result, the bandwidth-intensive GPU's requests in the oldest batch are prioritized over all CPU requests that are not in the oldest batch. This leads to significant CPU performance and fairness loss compared to SMS, which keeps the slowdown of CPU applications in check by employing the SJF batch scheduling policy described in Section 4.

ATLAS ranks applications based on their attained memory service and prioritizes higher-ranked (low-attained-service) applications over lower-ranked ones. As a result, it effectively deprioritizes the GPU, which has high memory bandwidth demand, for most of the time. ATLAS prioritizes the GPU over a CPU application only when the attained service of that application becomes higher than that of the GPU. While this attained-service based behavior of ATLAS was shown to lead to high unfairness in CPU-only systems (due to its almost-consistent deprioritization of heavily memory-intensive applications) [14], we found that ATLAS provides the best fairness (and the best performance) among previous algorithms because of the shortcomings of TCM we describe next. However, as it consistently deprioritizes the GPU and has a strict rank order between applications, ATLAS reduces the row-buffer locality of the GPU and leads to unfairness among CPU applications. SMS overcomes these two problems by 1) servicing GPU requests in batches, and 2) prioritizing requests at the batch level instead of enforcing a rank order among applications (which improves fairness).

TCM was shown to provide the best performance and fairness in CPU-only systems [15]. Grouping applications into latency-sensitive or bandwidth-sensitive clusters based on their memory intensities and employing a throughput oriented scheduling policy for the former cluster and a fairness oriented policy for the latter leads to both high system performance and fairness in CPU-only systems. However, in an integrated CPU-GPU system, the GPU generates a significantly larger number of memory requests compared to the CPU cores,

[7] In a 16-core CPU/1-GPU design, SMS requires sixteen 10-bit counters, sixteen 5-bit counters, and one 10-bit counter. In our experiment, we reset every 10,000 cycles.

Parameter	Setting	Parameter	Setting	Parameter	Setting	Parameter	Setting
CPU Clock Speed	3.2GHz	GPU Clock Speed	800Mhz	Channels/Ranks/Banks	4/1/8	DRAM Row buffer size	2KB
CPU reorder buffer	128 entries	GPU Max Throughput	1600 ops/cycle	DRAM Bus	64 bits/channel	tRCD/tCAS/tRP	12.5/12.5/12.5 ns
CPU L1 cache	32KB Private, 4-way	GPU Texture/Z/Color units	80/128/32	tRAS/tRC/tRRD	35/47.5/6.25 ns	tWTR/tRTP/tWR	7.5/7.5/15 ns
CPU L2 cache	8MB Shared, 16-way			CPU Cache Rep. Policy	LRU	MC request buffer size	300

Table 2. Simulation parameters.

which makes it difficult for TCM to appropriately group applications into two clusters using a single threshold value. As a result, TCM ends up grouping the GPU together with other memory-intensive CPU applications in the bandwidth-sensitive cluster. The fairness-oriented policy employed in this cluster causes the GPU to be prioritized over the memory-intensive CPU applications, thereby leading to high slowdowns for the deprioritized CPU applications and thus low fairness. In addition, we found that TCM makes non-robust clustering decisions, which classify some applications with high memory-intensity into the latency-sensitive cluster, again due to the fact that the intensity of the GPU skews the bandwidth measurements TCM uses to form the clusters. These misclassified memory-intensive applications that are admitted to the latency-sensitive cluster cause significant slowdowns to memory-non-intensive applications, leading to high unfairness and low system performance compared to ATLAS and SMS.

In contrast to all these previous schedulers, SMS prioritizes shorter batches of row-buffer-hit requests over longer batches (as determined by the SJF probability, p). As a result, SMS 1) preserves row-buffer locality, and most of the time, prioritizes the CPU applications which have shorter batches than the GPU (for sufficiently high values of p), leading to high system performance, and 2) does not result in a strict ranking of CPU applications among each other, leading to high fairness. As we will quantitatively demonstrate in Section 6, SMS provides the best CPU performance and fairness compared to these previous schedulers, while maintaining GPU frame rate above an acceptable level, at a lower hardware cost and with simpler scheduling logic.

5 Experimental Methodology

We use an in-house cycle-level simulator to perform our evaluations. For our performance evaluations, we model a system with sixteen x86-like CPU cores and a GPU. Each CPU core is a three-wide out-of-order processor with a cache hierarchy including per-core L1 caches and a shared L2 cache. The GPU does not share the CPU caches. The specification of the GPU we model is similar to the AMD Radeon 5870 specification [1]. The GPU has 20 cores, and each core consists of 16 SIMD functional units. A SIMD functional unit can process a VLIW instruction that can span up to 5 scalar operations, every cycle. As a result, the GPU has the capability to provide a maximum throughput of 1600 scalar operations per cycle. Table 2 shows the detailed system parameters for the evaluated CPU and GPU.

The parameters for the main memory system (DDR3 SDRAM-1600 [20]) are shown in Table 2. Unless stated otherwise, we use four memory controllers (one channel per memory controller) for all experiments. In order to prevent the GPU requests from occupying the vast majority of the request buffer entries, we reserve half of the request buffer entries for CPU requests for non-SMS MCs. To model the memory bandwidth used by the GPU accurately, we perform coalescing of GPU memory requests before they are sent to the MC as is done in current GPUs [17].

Workloads: We evaluate our system with a set of 105 multi-programmed workloads, each simulated for 500 million cycles. Each workload consists of sixteen SPEC CPU2006 benchmarks and one GPU application selected from a mix of video games and graphics performance benchmarks described in Table 3. For each CPU benchmark, we use Pin [18] with PinPoints [29] to select the representative phase. Our GPU traces are collected from GPU benchmarks and recent games through a proprietary GPU simulator. Note that our GPU memory traces include memory requests coming from fixed function units (FFUs). Table 3 shows the memory intensity of GPU applications, measured in terms of misses per kilo instructions (MPKI)[8] and also the percentage of memory requests that come from fixed function units. GPU memory accesses are collected after having first been filtered through the GPU's internal cache hierarchy; therefore, we do not further model internal GPU caches in our hybrid CPU-GPU simulation framework.

Name	Description	MPKI	% of Requests from FFUs
Bench01	3D Mark 1	204.0	62.3%
Bench02	3D Mark 2	267.0	54.2%
Bench03	3D Mark 3	419.7	99.6%
Game01	Shooting Game 1	304.7	80.2%
Game02	Shooting Game 2	173.5	51.0%
Game03	Shooting Game 3	345.8	60.3%
Game04	Adventure Game	206.6	61.0%
Game05	Role-playing Game	321.2	78.5%

Table 3. GPU benchmarks.

We classify CPU benchmarks into three categories (Low, Medium, and High) based on their memory intensities, measured as last-level cache misses per thousand instructions (MPKI). Table 4 shows the MPKI for each CPU benchmark. Benchmarks with less than or equal to 1 MPKI are classified as low memory intensity, between 1 and 15 MPKI as medium memory intensity, and greater than 15 as high memory intensity. Based on these three categories, we randomly choose a number of benchmarks from each category to form workloads with seven different intensity mixes: L (All Low), ML (Medium/Low), M (All Medium), HL (High/Low), HML (High/Medium/Low), HM (High/Medium) and H (All High). The co-running GPU benchmark is randomly selected for each workload.

Name	MPKI	Name	MPKI	Name	MPKI
povray	0.01	gobmk	0.51	milc	16.18
tonto	0.01	gromacs	1.12	xalancbmk	18.32
calculix	0.04	h264ref	1.22	omnetpp	19.20
perlbench	0.08	hmmer	2.82	GemsFDTD	24.70
dealII	0.11	bzip2	4.10	libquantum	26.24
namd	0.11	cactusADM	4.74	soplex	27.96
wrf	0.12	astar	5.19	lbm	28.30
gcc	0.22	sphinx3	13.21	mcf	74.35
sjeng	0.37	leslie3d	14.79		

Table 4. L2 Cache Misses Per Kilo-Instruction (MPKI) of 26 SPEC 2006 benchmarks.

Performance Metrics: We measure the performance of the multi-core CPU using the weighted speedup metric [8, 33], as shown in Equation 1. We measure the performance of the GPU by calculating the frame rate (frames per second), a commonly used graphics performance metric [16, 32]. We compute the GPU speedup as the ratio of its frame rate when it is sharing the system with the CPU over the GPU's frame rate when it is run alone on the same baseline system, as shown in Equation 2.

$$CPU_{WS} = \sum_{i=1}^{NumCores} \frac{IPC_i^{shared}}{IPC_i^{alone}} \quad (1) \qquad GPU_{Speedup} = \frac{GPU_{FrameRate}^{shared}}{GPU_{FrameRate}^{alone}} \quad (2)$$

Measuring overall system performance in an integrated CPU-GPU system is not straightforward. Using the conventional weighted speedup metric for this purpose weights the GPU as equivalent to a single CPU core. However, having the same weight for the GPU as a single CPU core does not allow for flexibility in quantifying the system performance of different systems that have different performance requirements. In some systems (e.g., those used by video gaming enthusiasts), GPU performance could be much more important than CPU performance

[8]MPKI of a GPU application is calculated as the total number of memory accesses divided by the number of scalar GPU operations, where a scalar GPU operation is the closest unit of work to a CPU instruction.

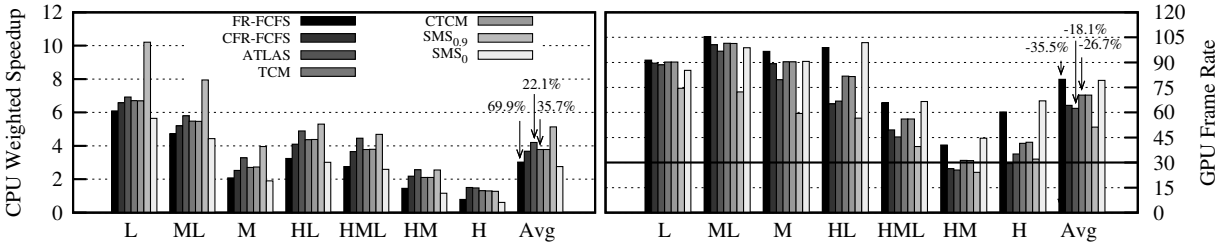

Figure 5. Performance of CPU and GPU for 7 workload categories (105 workloads). % values are gains of $SMS_{0.9}$ over FR-FCFS, ATLAS, and TCM.

to the end-user. In contrast, in some other systems (e.g., those with interactive applications running together with graphics or in mobile systems), CPU performance could be much more important than GPU performance. Therefore, we need a metric that is flexible to evaluate the performance of different heterogeneous systems, given different end-user needs. To this end, we propose an integrated CPU-GPU Weighted Speedup (CGWS) metric (Eqn. 3) where the GPU's speedup is weighted by a parameter GPU_{weight}.

$$CGWS = CPU_{WS} + GPU_{weight} * GPU_{Speedup} \quad (3)$$

The GPU_{weight} parameter determines the weight of the GPU relative to a CPU core. When the GPU_{weight} is set to 1, the GPU is treated as equivalent to a single CPU core, and CPU performance (weighted speedup) becomes an important factor in determining system performance. On the other hand, when the GPU_{weight} is set to a large value (e.g., 1000), CPU performance becomes negligible, and GPU performance becomes the dominant factor. Section 6.2.1 shows the performance of our system for a wide range of values of the GPU_{weight} parameter.

In addition to system performance, we measure *Unfairness* (Eqn. 4) using maximum slowdown [5, 14, 15, 34] over all of the CPU cores and the GPU. We report the harmonic mean instead of arithmetic mean for *Unfairness* in our evaluations because slowdown is the inverse metric of speedup.

$$Unfairness = \max \left\{ \max_i \frac{IPC^{alone}_{Core_i}}{IPC^{shared}_{Core_i}}, \frac{GPU^{alone}_{FrameRate}}{GPU^{shared}_{FrameRate}} \right\} \quad (4)$$

Parameters of Evaluated Algorithms: For SMS, we set the *batching FIFO queue* size for each CPU core to 10 entries, and for the GPU to 20 entries. The *DCS FIFO* queue size for each bank is 15 entries. In addition to the *age thresholds* given in Section 4.2, the thresholds for a low-intensity CPU application ($<$ 1 MPKC) and the GPU are 0 and 800 cycles, respectively. We use a *QuantumLength* of 1M cycles and HistoryWeight of 0.875 for ATLAS. We use a *ClusterThresh* of 0.1 and *ShuffleCycles* of 800 for TCM.

Area and Power Modeling: We use Verilog models synthesized with a commercial 180nm design library. The standard-cell synthesis models area, leakage power and timing of FR-FCFS and SMS scheduling logic and buffers.[9] In Section 6.4, we report the area and leakage power of SMS normalized to FR-FCFS.

6 Experimental Evaluation

6.1 Individual CPU and GPU Performance

First, we compare the individual CPU and GPU performance of SMS with three state-of-the-art memory scheduler configurations: FR-FCFS, ATLAS and TCM, and their variants, CFR-FCFS and CTCM, which are similar to FR-FCFS and TCM respectively, but always prioritize the CPU cores' requests over the GPU's requests. We present SMS with two values of p, the SJF probability: $SMS_{0.9}$ denotes the configuration where $p = 0.9$ (i.e., with 90% probability the application with fewest total requests is selected to be scheduled), and SMS_0 denotes the configuration where $p = 0$ (i.e., the batch scheduler always is round-robin). Results are presented across 105 workloads of 7 intensity categories described in Section 5, with workload memory intensities increasing from left to right.

Figure 5 shows the performance of the CPU (weighted speedup), and the individual performance of the GPU (frame rate) separately. Three major conclusions are in order. First, $SMS_{0.9}$ improves CPU performance by 22.1%/ 35.7% over the two best previous scheduling policies, ATLAS and TCM, on average. This is because ATLAS, TCM, and other previously proposed scheduling policies[10] are not designed to robustly handle the high degree of disparity in memory access intensities present between CPU and GPU applications in an integrated CPU-GPU system, as we explained in Section 4.5. Previous scheduling policies allocate a large fraction of the system memory bandwidth to the GPU, thereby penalizing the CPU applications. Second, CFR-FCFS provides better CPU performance than FR-FCFS as it prioritizes CPU requests, preventing the GPU's row-buffer hits from hogging memory bandwidth. CTCM performs similarly as TCM because the baseline TCM is already application-aware as it prioritizes low-intensity applications over others. However, $SMS_{0.9}$ still provides better CPU performance than both CFR-FCFS and CTCM, on average across all workloads. Third, $SMS_{0.9}$ achieves this CPU performance improvement over previous schedulers and their variants by appropriately restricting the fraction of system bandwidth allocated to the GPU, resulting in 18.1%/ 26.7% reduction in GPU frame rate as compared to ATLAS and TCM, on average. Although $SMS_{0.9}$ reduces the frame rate of the GPU, it still maintains a frame rate greater than 30 frames per second (for all workload categories except the HM category), which is likely acceptable for a large number of applications and users. In systems and use cases that require a higher GPU frame rate, we can adjust the value of p, the SJF probability, to ensure that an adequate frame rate is achieved. In particular, Figure 5 shows that SMS_0 ($p = 0$) provides a significantly higher GPU frame rate, at the cost of lower CPU performance. In Section 6.2.1, we present further analysis on the tradeoff between CPU and GPU performance by configuring the SJF probability appropriately.

6.2 Combined CPU-GPU Performance

6.2.1 Effect of Configurable SJF Probability

In this section, we present and analyze the combined CPU-GPU performance of the 16-core CPU/1 GPU system and the effect of changing the parameter p (SJF probability).

Figure 6 (a) shows the performance of SMS at different values of p, the SJF probability, when the GPU_{weight} is varied from 0.001 to 1000. This weight represents the relative importance of GPU performance vs. CPU performance in the combined CPU-GPU Weighted Speedup (CGWS) metric. Two key conclusions are in order. First, $SMS_{0.9}$ has higher CGWS than FR-FCFS, ATLAS and TCM, for low values of GPU_{weight} (< 7.5). However, as GPU_{weight} increases, $SMS_{0.9}$ has lower CGWS than previous scheduling policies like FR-FCFS, ATLAS and TCM. This is because previous scheduling policies prioritize the GPU at varying levels, while penalizing CPU cores, hence providing higher GPU performance, as illustrated in Figure 5 and de-

[9] Each buffer is modeled using a flip-flop. Neither logic nor buffers are optimized for area and power.

[10] We do not show PAR-BS in these figures so as not to clutter the graphs, but it consistently has lower performance and fairness than that of ATLAS/TCM.

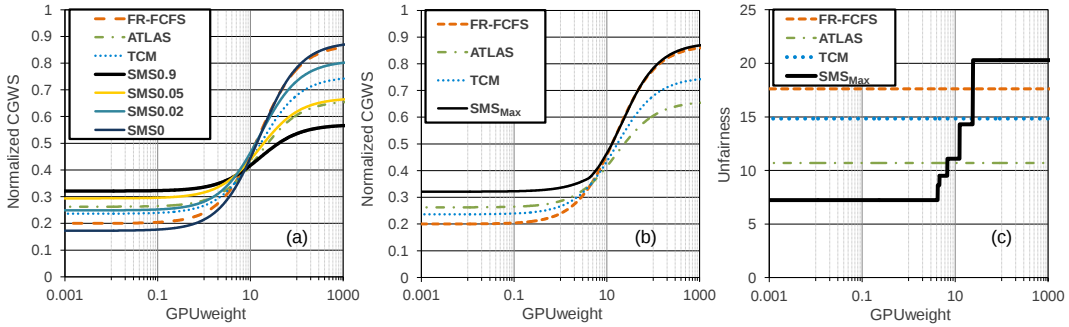

Figure 6. Combined performance of the CPU-GPU system (a) for different SMS configurations with different SJF probability, (b) for the SMS configuration that provides the maximum CGWS, and (c) unfairness for the configuration that provides the maximum CGWS.

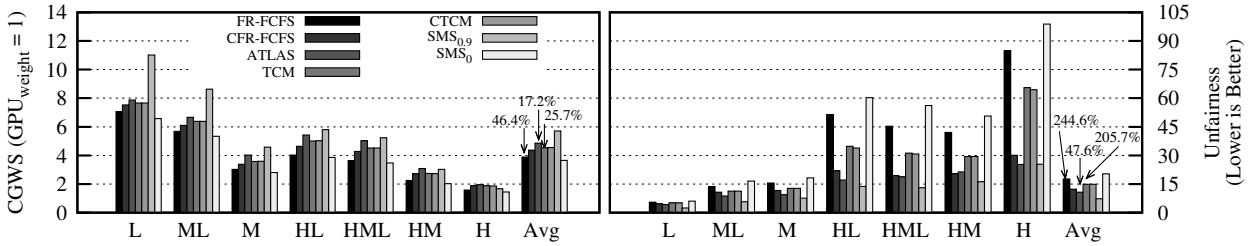

Figure 7. System performance and fairness for 7 workload categories (105 workloads). % values are gains of $SMS_{0.9}$ over FR-FCFS, ATLAS, and TCM.

scribed qualitatively in Section 4.5. Therefore, they provide higher CGWS than SMS, when the contribution of the GPU to the CGWS metric is weighted significantly higher. Second, although $SMS_{0.9}$ provides a lower value of CGWS compared to previous algorithms when GPU_{weight} is higher than 7.5, SMS_0 leads to higher CGWS than all previous algorithms. A small p value enables SMS to prioritize GPU requests much more often than CPU requests (due to round-robin scheduling of batches and long batches for the GPU), thereby largely improving GPU performance over other algorithms except FR-FCFS (which achieves a similar effect by prioritizing the row hit requests, which are much more common in the GPU than in the CPU).

Note that p is an adjustable knob provided by our SMS design, which can be statically or dynamically configured to trade off between CPU and GPU performance, depending on the needs of the system. If a designer or system software care more about GPU performance, they can set p to be very low (causing the SMS algorithm to implicitly favor the GPU over the CPU, as we have seen in Figure 6(a)). On the other hand, if they care more about CPU performance, they can set p to be very high (causing the SMS algorithm to favor the CPU cores over the GPU). Therefore, one can determine and select at design time the best SJF probability, p for a given GPU_{weight}. This means that for each GPU_{weight}, we can find an SJF probability that maximizes CGWS.[11]

Figures 6 (b) and (c) show the performance and fairness of SMS_{Max}, which is defined as SMS with a p value that maximizes CGWS for each given GPU_{weight}. Two key observations are in order. First, as we increase the GPU_{weight}, a p value that maximizes CGWS favors the GPU over CPU cores, increasing the slowdown of CPU applications, and thus leading to higher unfairness. However, for smaller values of GPU_{weight} (i.e., when CPU performance is important), SMS provides the highest fairness across all schedulers. Second, SMS_{Max} provides the best CPU-GPU Weighted Speedup (CGWS) compared to FR-FCFS, ATLAS and TCM across all values of GPU_{weight} considered. Therefore, we conclude that SMS can be configured to consistently provide better system performance than state-of-the-art memory schedulers regardless of how important the CPU or the GPU is to system performance.

[11] In fact, new instructions can be added to the ISA to change p dynamically to achieve the best performance in cases where GPU_{weight} (importance of the GPU) varies dynamically.

6.2.2 Performance & Fairness Analysis (GPU_{weight} = 1)

We study the system performance and fairness of SMS when the GPU_{weight} has a low value of 1. This represents a system or a dynamic use case where CPU performance matters significantly to the user, e.g., a typical system running a GPU application alongside multiple CPU applications.

Figure 7 shows the system performance (measured as CGWS) and fairness of the previously proposed algorithms, $SMS_{0.9}$, and SMS_0 for all workload categories, when the GPU_{weight} is 1. $SMS_{0.9}$ provides 46.4%/17.2%/25.7% system performance improvement and 244.6%/47.6%/205.7% fairness improvement over FR-FCFS/ATLAS/TCM. SMS_0 provides 6.8%/33.4%/24.3% system performance degradation and 15.2%/89.7%/36.7% fairness degradation over FR-FCFS/ATLAS/TCM, respectively. As discussed in Section 4, $SMS_{0.9}$ improves system performance and fairness for a low GPU_{weight} of 1, as it prioritizes the short batches of the CPU cores over the long batches of the GPU, while SMS_0 degrades system performance, as it selects between CPU and GPU batches in merely a round-robin fashion. Therefore, we conclude that for a GPU_{weight} of 1, SMS with an appropriately selected p value of 0.9 provides better system performance and fairness than all previously proposed scheduling policies.

Based on the results for each workload category, we make the following major observations. First, $SMS_{0.9}$ consistently outperforms previously proposed algorithms both in terms of system performance and fairness across most of the workload categories. Second, in the "H" category with only high memory-intensity applications, $SMS_{0.9}$ underperforms ATLAS and TCM by 16.7% and 12.5%, but it still provides 7.1% higher system performance than FR-FCFS. ATLAS' attained service based scheduling tends to prioritize all CPU applications equally, as all CPU applications in a "H" category workload have similar memory intensities. This leads to high performance and fairness for ATLAS (SMS achieves similar fairness). TCM, on the other hand, improves performance by unfairly prioritizing certain applications over others, which is reflected by its higher unfairness in the "H" category, as compared to other workload categories. Specifically, TCM misclassifies some high memory-intensity applications into the low memory-intensity cluster, which starves the requests of applications in the high memory-intensity cluster. On the other hand, SMS preserves fairness in all workload categories by using its probabilistic SJF batch scheduling policy described in Section 4. As a result, SMS provides 205.7% better fairness relative to TCM.

6.2.3 Performance & Fairness Analysis (GPU$_{weight}$=1000)

We also study the system performance and fairness of SMS when the GPU$_{weight}$ is 1000. This represents a system or a use case where CPU performance matters less to the user than GPU performance does, e.g., a system intended for gaming or watching high definition video. Figure 8 shows the system performance (measured as CGWS) of the previously proposed algorithms, SMS_0, and $SMS_{0.9}$ for all workload categories when the GPU$_{weight}$ is 1000. SMS_0 provides 1.6%/32.7%/16.4% system performance improvement, while $SMS_{0.9}$ provides 44.3%/13.3%/27.7% system performance degradation, over FR-FCFS/ATLAS/TCM. Fairness of these algorithms was already shown in Figure 7 (note that the unfairness measure does not depend on GPU$_{weight}$). However, when GPU$_{weight}$ is high, fairness is not an important concern because the goal is to optimize for GPU performance. Two key observations are in order from Figure 8. First, when the GPU$_{weight}$ is very high, FR-FCFS, which predominantly prioritizes the GPU owing to the GPU's very high row-buffer locality and very high memory intensity, provides the best system performance among previous techniques. Second, SMS_0 provides similar system performance as FR-FCFS, on average, across all workload categories, as it predominantly prioritizes the GPU requests when a low value of p is used. Therefore, we conclude that for a GPU$_{weight}$ of 1000, SMS with an appropriately selected p value of 0 provides better system performance than previously proposed scheduling policies.

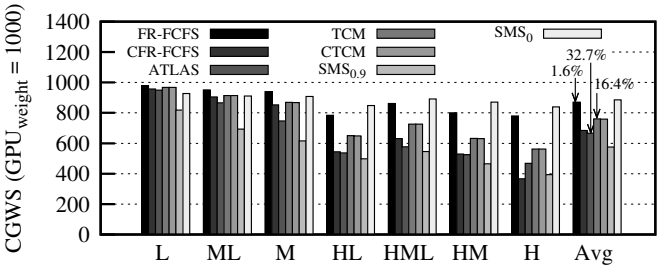

Figure 8. System performance for 7 workload categories (105 workloads). % values are gains of SMS_0 over FR-FCFS, ATLAS, and TCM.

Observing the results for each workload category, we see that SMS_0 performs better as the CPU workload intensity increases. In the high-intensity workloads that have high interference between the CPU and GPU, SMS_0 allows the GPU to form batches with high row-buffer locality without being interfered by requests from high-intensity CPU applications. On the other hand, FR-FCFS performs better at lower CPU workload intensities. When the CPU applications in a workload have low memory intensities, they seldom interfere with the GPU application. Thus, merely prioritizing row-buffer hits favors the GPU application, which has high row-buffer locality, thereby providing high GPU performance (which is good in this scenario as GPU$_{weight}$ is 1000). However, the wait time for batch formation in SMS_0 can sometimes throttle back the GPU application unnecessarily, leading to a slight performance degradation for the GPU. On the other hand, when the CPU applications in a workload have high memory intensities, these applications sometimes interfere with the GPU application in the row buffers when a naive application-unaware scheduling policy like FR-FCFS is used. In contrast, SMS explicitly prioritizes the GPU application by using a very low value of the parameter p, which improves GPU's row buffer locality, leading to a larger improvement in GPU performance and, thus, CGWS than FR-FCFS.

6.3 Scalability with CPU Cores and Memory Channels

Figure 9 compares the CPU performance (weighted speedup), GPU performance (frame rate), and fairness of SMS (we use $SMS_{0.9}$ for our sensitivity studies) against FR-FCFS/ATLAS/TCM (averaged over 75 workloads[12]) with the same number of request buffers, as the number of CPU cores is varied. We make the following observations. First, SMS always provides better CPU performance and fairness than FR-FCFS/ATLAS/TCM. Second, the fairness gains increase as the number of CPU cores increases to 16, as this significantly reduces the memory bandwidth per core given the same memory system configuration. With lower bandwidth per core, the interference between applications becomes more severe. SMS mitigates the interference by effectively throttling back the GPU. Third, similar to the observation made in Section 6.1, SMS reduces GPU frame rate compared to other memory schedulers. However, SMS still maintains a frame rate above 30 frames per second for all systems. In general, SMS provides consistent improvement in CPU performance and fairness relative to other memory schedulers while maintaining likely acceptable GPU performance (30 frames per second) across systems with different numbers of CPU cores. SMS' benefits are likely to become more significant as CPU core counts in future technology nodes increase, as this would lead to higher interference between the CPU cores and the GPU, which previous scheduling algorithms like ATLAS and TCM do not handle robustly.

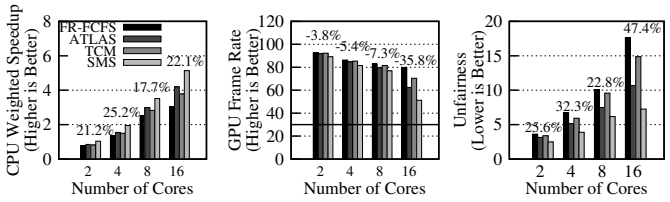

Figure 9. Performance and fairness vs. number of CPU cores. % values are gains of $SMS_{0.9}$ over the best previous algorithms (ATLAS for weighted speedup and fairness, and FR-FCFS for GPU frame rate).

Figure 10 shows the CPU performance (weighted speedup), GPU performance (frame rate), and fairness of *SMS* compared against FR-FCFS, ATLAS, and TCM as the number of memory channels is varied. Three key observations are in order. First, *SMS* provides consistent improvement in CPU performance and fairness across a wide range of system configurations with different numbers of channels. Second, as the number of memory channels increases to eight, the available memory bandwidth increases, and hence SMS is able to provide GPU frame rates that are within 13.4%/7.6%/10.5% of FR-FCFS/ATLAS/TCM, while improving CPU system performance by 54.7%/22.2%/38.5%, respectively. Third, the GPU frame rate of *SMS* drops below 30 frames per second in a two-channel system. However, these results are without an appropriate reconfiguration of p. As described in Section 6.2.1, *SMS* can provide more adequate GPU frame rates by adjusting the SJF probability, p, while trading off CPU performance. We conclude that SMS provides high CPU performance and high fairness compared to previously proposed scheduling algorithms without degrading the GPU frame rate below a usually acceptable level, across a wide range of system configurations with different core and memory channel counts.

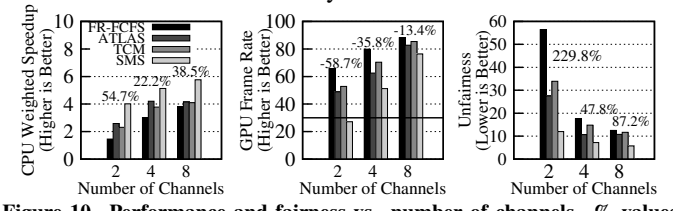

Figure 10. Performance and fairness vs. number of channels. % values are gains of $SMS_{0.9}$ over the best previous algorithms (ATLAS for weighted speedup and fairness, and FR-FCFS for GPU frame rate).

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

6.4 Power and Area

We present the power and area of FR-FCFS and SMS in Table 5 for a system with 300 request buffers. Two major conclusions are in order. First, SMS consumes 66.7% less leakage power than FR-FCFS, the simplest of all evaluated previous memory schedulers. Second, SMS requires 46.3% less area than FR-FCFS. The majority of the power and area savings of SMS over FR-FCFS comes from the decentralized request buffer queues and simpler scheduling logic in SMS, as compared to centralized request buffer queues, CAMs, and complex scheduling logic in FR-FCFS. Because ATLAS and TCM require more complex ranking and scheduling logic than FR-FCFS does, SMS provides likely more significant power and area reductions over ATLAS and TCM.

Data Type	FR-FCFS	SMS
Leakage (Normalized)	1	0.667
Area (Normalized)	1	0.463

Table 5. Power and area comparison.

6.5 Analysis of SMS Algorithm Parameters

Figure 11 shows CPU performance, fairness and GPU frame rate (all normalized to $SMS_{0.9}$), when we vary *threshold age* (left) for applications in all intensity categories from 0% (no batching) to 300% of the default values specified in Section 5 and *DCS FIFO queue size* (right) from 5 to 20 entries. Three observations are in order. First, as *threshold age* increases, the GPU frame rate increases. However, CPU performance degrades, when *threshold age* is increased beyond a point. This is because as *threshold age* increases, SMS takes longer to form a batch. While longer batches help improve the GPU's row-buffer hit rate and hence its frame rate, they increase the wait time of latency-sensitive CPU applications, degrading CPU performance. Second, as *threshold age* decreases, batches become smaller, degrading row-buffer locality. This reduces both GPU performance and CPU performance; however, GPU performance degrades more than CPU performance because $SMS_{0.9}$ prioritizes CPU requests over GPU requests, leading to additional interference from the CPU cores to the GPU. Third, CPU performance and fairness increase while GPU performance decreases as we lower *DCS FIFO queue size*. When *DCS FIFO queue size* becomes smaller, the DCS FIFOs can hold a smaller number of batches. Hence, once a few batches are scheduled to a DCS FIFO, other batches would need to wait until the batches in the DCS FIFO are drained. This particularly impacts the GPU, as $SMS_{0.9}$ prioritizes CPU requests over GPU requests. This reduction in GPU performance leads to lower interference from the GPU, and hence increases CPU performance. We also found that increasing the DCS FIFO size beyond 20 entries provides only very small performance benefit.

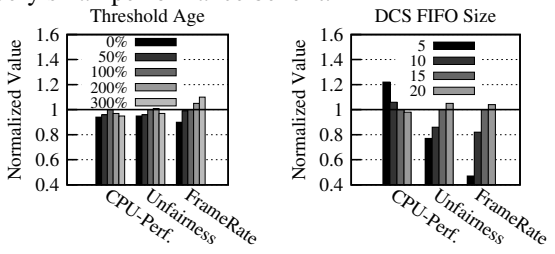

Figure 11. Sensitivity of SMS to different design parameters.

6.6 Analysis of CPU-only Systems

In Sections 6.1-6.3, we showed that SMS effectively mitigates inter-application interference in a CPU-GPU integrated system. In this section, we evaluate the performance of SMS in a CPU-only scenario.[13] Figure 12 shows the system performance and fairness of SMS on a 16-core system with exactly the same system parameters as described in Section 5, except that the system does not have a GPU. We present results for only workload categories with at least some high memory-intensity applications, as the performance and fairness of SMS in the other workload categories are quite similar to that of ATLAS and TCM. First, we observe that SMS degrades performance by only 8.8% and 4% respectively compared to ATLAS and TCM, while it improves fairness by 1.2% and 25.7% respectively, on average across all workloads, compared to ATLAS and TCM. Compared to ATLAS, SMS improves fairness in "HL", "HML" and "HM" categories but reduces the performance in those categories slightly. However, ATLAS provides the best fairness in the "H" workload category, as ATLAS' attained service based prioritization mechanism achieves the same effect as shuffling request priorities between applications, when they have similar memory intensities. Compared to TCM, SMS' performance degradation mainly comes from the "H" workload category; as discussed in our main evaluations in Section 6, TCM places some high memory-intensity applications into the low memory-intensity cluster, which results in the strict prioritization of such applications over those (slightly more intensive applications) placed in the high memory-intensity cluster. Therefore, TCM gains performance at the cost of unfairness to the memory-intensive applications that fall into the latter cluster. On the other hand, SMS reduces unfairness with its probabilistic SJF policy, while still maintaining good system performance. We conclude that SMS can be employed in CPU-only systems as an effective, low-complexity, and scalable memory scheduling policy to provide high fairness and reasonably high performance.

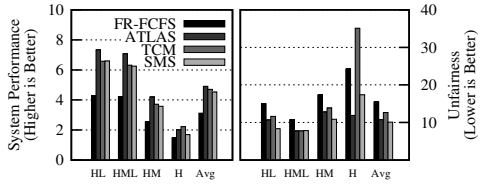

Figure 12. Results on a 16-core CPU system for 4 workload categories.

7 Related Work

Memory Scheduling: We have already compared SMS to state-of-the-art memory schedulers designed for CPU-only systems (FR-FCFS [31, 37], ATLAS [14], PAR-BS [25], and TCM [15]). SMS outperforms these schedulers in terms of both system performance and fairness as explained in Sections 4.5 and 6. Other memory schedulers (e.g., [11, 19, 21, 22, 24, 26, 30, 35, 36]) that are either aware or unaware of applications have been proposed for CPU-only systems. However, none of these works has been shown to provide better performance and fairness than TCM, which SMS outperforms across a wide range of workloads. Furthermore, previous works [2, 3, 35] have also explored memory scheduling for GPUs/GPGPUs. However, these works are not targeted toward integrated CPU-GPU systems and are application-unaware. On the other hand, SMS is a new application-aware memory scheduler that mitigates interference between CPU and GPU applications in heterogeneous CPU-GPU systems.

Parallel Application Memory Scheduling: Ebrahimi et al. [7] propose a memory scheduler to explicitly manage inter-thread memory interference in parallel applications. However, the proposed memory scheduler is not designed to mitigate interference between heterogeneous applications in CPU-GPU systems. Their proposed principles can be combined with our work to prioritize *limiter threads* in CPU-GPU systems. We aim to investigate the interaction between parallel CPU applications and GPU applications as part of future work.

QoS for CPU-GPU heterogeneous systems: A concurrent work by Jeong et al. proposes a QoS-aware memory controller that has the explicit goal of providing QoS for the GPU by adjusting the priority of the CPU and the GPU based on whether

[13] In view of simulation bandwidth and time constraints, we reduce the simulation time to 300M cycles for these studies. We did not tune the configuration parameters of ATLAS, TCM and SMS for CPU-only cases.

the GPU meets its target frame rate [12]. Our goal, in contrast, is to provide high system performance and fairness, which makes our approach complementary. Principles from [12] can be applied to SMS to dynamically adjust the priority of the GPU (p value), which can provide QoS to the GPU and further improve system performance.

Sub-row Interleaving: In order to preserve row-buffer locality and prevent applications from monopolizing the row buffer when an open-page policy is used, Kaseridis et al. [13] propose a scheme that interleaves data at a sub-row granularity across channels and banks. Our batch formation process provides similar qualitative benefits to the sub-row interleaving approach of limiting row-buffer hit streaks. However, their proposal still needs a large centralized scheduler that requires complex logic to support the memory access demands of the vastly different CPU and GPU applications.

Memory Channel Partitioning: Instead of mitigating memory interference between applications by intelligently prioritizing requests at the memory controller, Muralidhara et al. [23] propose memory channel partitioning (MCP) to map the data of badly interfering applications to different channels. Our approach is complementary to MCP, as it has been shown that MCP can be integrated with different memory schedulers to further improve system throughput and fairness [23].

Source Throttling: Ebrahimi et al. [6] mitigate inter-application interference in a shared-memory system by throttling the request injection rates of interference-causing applications at the source. This approach is complementary to SMS which reduces inter-application interference via memory scheduling. Note that, the staged nature of SMS allows throttling of sources at the source FIFOs.

8 Conclusion

While many advances in memory scheduling policies have been made to deal with multi-core processors, the integration of GPUs on the same chip as CPUs has created new system design challenges. We demonstrate how the inclusion of GPU memory traffic can cause severe difficulties for existing memory controller designs in terms of performance, fairness and design complexity. To solve this problem, we propose a fundamentally new approach to memory scheduler design, the Staged Memory Scheduler (SMS). The key idea of SMS is to decouple the primary functions of an application-aware memory controller into three stages: (1) per-source FIFO queues that group requests to the same row into *batches* for each source/application, to improve row-buffer locality, (2) the batch scheduler that prioritizes between the batches of different sources, to maximize performance and fairness and (3) the DRAM command scheduler that simply issues requests to DRAM in the order it receives them (as the previous stages have already made application-aware and row-buffer locality aware batch scheduling decisions). This staged design provides two benefits. First, it is simpler and more scalable than previous application-aware memory schedulers with large monolithic request buffers. Second, it provides a new memory scheduling algorithm by composing these three separate stages. Our evaluations show that SMS delivers better performance and fairness compared to state-of-the-art memory schedulers, while providing a design that is significantly simpler to implement. We conclude that the new staged approach to memory scheduling can provide a scalable substrate for memory controller design for future heterogeneous systems.

Acknowledgments

We thank Stephen Somogyi and Fritz Kruger at AMD for their assistance with the modeling of the GPU applications. We also thank Antonio Gonzalez, anonymous reviewers and members of the SAFARI group at CMU for their feedback. We acknowledge the generous support of AMD, Intel, Oracle, and Samsung. This research was also partially supported by grants from NSF (CAREER Award CCF-0953246 and CCF-1147397), GSRC, and Intel ARO Memory Hierarchy Program. Rachata Ausavarungnirun is supported by the Royal Thai Government Scholarship, and was an intern at AMD while he did part of this work.

References

[1] Advanced Micro Devices. *AMD Radeon HD 5870 Graphics*. http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870.
[2] N. B Lakshminarayana et al. DRAM scheduling policy for GPGPU architectures based on a potential function. *IEEE CAL*, 2011.
[3] N. B. Lakshminarayana and H. Kim. Effect of instruction fetch and memory scheduling on GPU performance. *GPGPU Workshop*, 2010.
[4] B. Burgess et al. Bobcat: AMD's low-power x86 processor. *IEEE Micro*, 2011.
[5] R. Das et al. Application-aware prioritization mechanisms for on-chip networks. In *MICRO-42*, 2009.
[6] E. Ebrahimi et al. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *ASPLOS-15*, 2010.
[7] E. Ebrahimi et al. Parallel application memory scheduling. In *MICRO-44*, 2011.
[8] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. In *MICRO-41*, 2008.
[9] A. Fog. *The Microarchitecture of Intel, AMD and VIA CPUs*. http://www.agner.org/optimize/#manuals.
[10] Intel. *Sandy Bridge Intel processor graphics performance developer's guide*. http://software.intel.com/file/34436.
[11] E. Ipek et al. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.
[12] M. K. Jeong et al. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *DAC-49*, 2012.
[13] D. Kaseridis et al. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *MICRO-44*, 2011.
[14] Y. Kim et al. ATLAS: a scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA-16*, 2010.
[15] Y. Kim et al. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *MICRO-43*, 2010.
[16] V. W. Lee et al. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA-37*, 2010.
[17] E. Lindholm et al. NVidia Tesla: A unified graphics and computing architecture. *IEEE Micro*, 2008.
[18] C. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI, 2005*.
[19] S. A. McKee et al. Dynamic access ordering for streamed computations. *IEEE Transactions on Computers*, 2000.
[20] Micron Technology Inc. *1Gb: x4, x8, x16 DDR3 SDRAM Features*. `http://download.micron.com/pdf/datasheets/dram/ddr3/1Gb_DDR3_SDRAM.pdf`.
[21] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security*, 2007.
[22] T. Moscibroda and O. Mutlu. Multi-level DRAM controller to manage access to DRAM. *U.S. Patent Number 8001338*, Aug 2011.
[23] S. Muralidhara et al. Reducing memory interference in multi-core systems via application-aware memory channel partitioning. In *MICRO-44*, 2011.
[24] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO-40*, 2007.
[25] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA-35*, 2008.
[26] K. J. Nesbit et al. Fair queuing memory systems. In *MICRO*, 2006.
[27] NVIDIA. *Tegra 2 Tech. Ref. Manual*. http://www.nvidia.com/object/tegra-2.html.
[28] S. Palacharla et al. Complexity-effective superscalar processors. In *ISCA-24*, 1997.
[29] H. Patil et al. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.
[30] N. Rafique et al. Effective management of DRAM bandwidth in multicore processors. In *PACT*, 2007.
[31] S. Rixner et al. Memory access scheduling. In *ISCA-27*, 2000.
[32] C. J. Rossbach et al. PTask: operating system abstractions to manage GPUs as compute devices. In *SOSP-23*, 2011.
[33] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS-9*, 2000.
[34] H. Vandierendonck and A. Seznec. Fairness metrics for multi-threaded processors. *IEEE CAL*, 2011.
[35] G. L. Yuan et al. Complexity effective memory access scheduling for many-core accelerator architectures. In *MICRO-42*, 2009.
[36] Z. Zhu and Z. Zhang. A performance comparison of DRAM memory system optimizations for SMT processors. In *HPCA-11*, 2005.
[37] W. K. Zuravleff and T. Robinson. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. *U.S. Patent Number 5,630,096*, May 1997.