BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling

Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu

Abstract—In a multicore system, applications running on different cores interfere at main memory. This inter-application interference degrades overall system performance and unfairly slows down applications. Prior works have developed application-aware memory request schedulers to tackle this problem. State-of-the-art application-aware memory request schedulers prioritize memory requests of applications that are vulnerable to interference, by ranking individual applications based on their memory access characteristics and enforcing a total rank order. In this paper, we observe that state-of-the-art application-aware memory schedulers have two major shortcomings. First, such schedulers trade off hardware complexity in order to achieve high performance or fairness, since ranking applications individually with a total order based on memory access characteristics leads to high hardware cost and complexity. Such complexity could prevent the scheduler from meeting the stringent timing requirements of state-of-the-art DDR protocols. Second, ranking can unfairly slow down applications that are at the bottom of the ranking stack, thereby sometimes leading to high slowdowns and low overall system performance. To overcome these shortcomings, we propose *the Blacklisting Memory Scheduler (BLISS)*, which achieves high system performance and fairness *while* incurring low hardware cost and complexity. BLISS design is based on two new observations. First, we find that, to mitigate interference, it is sufficient to separate applications into only two groups, one containing applications that are vulnerable to interference and another containing applications that cause interference, instead of ranking individual applications with a total order. Vulnerable-to-interference group is prioritized over the interference-causing group. Second, we show that this grouping can be efficiently performed by simply counting the number of consecutive requests served from each application. We evaluate BLISS across a wide variety of workloads and system configurations and compare its performance and hardware complexity (via RTL implementations), with five state-of-the-art memory schedulers. Our evaluations show that BLISS achieves 5 percent better system performance and 25 percent better fairness than the best-performing previous memory scheduler while greatly reducing critical path latency and hardware area cost of the memory scheduler (by 79 and 43 percent, respectively), thereby achieving a good trade-off between performance, fairness and hardware complexity.

---

1 INTRODUCTION

IN modern systems, the high latency of accessing large-capacity off-chip memory and limited memory bandwidth have made main memory a critical performance bottleneck. In a multicore system, main memory is typically shared by applications running on different cores (or, hardware contexts). Requests from such applications contend for the off-chip memory bandwidth, resulting in interference. Several prior works [1], [2], [3], [4] demonstrated that this inter-application interference can severely degrade overall system performance and fairness. This problem will likely get worse as the number of cores on a multicore chip increases [2].

Prior works proposed different solution approaches to mitigate inter-application interference, with the goal of improving system performance and fairness (e.g., [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]). A prevalent solution direction is application-aware memory request scheduling (e.g., [3], [4], [5], [6], [7], [12]). The basic idea of application-aware memory scheduling is to

prioritize requests of different applications differently, based on the applications' memory access characteristics. State-of-the-art application-aware memory schedulers typically i) *monitor* applications' memory access characteristics, ii) *rank applications individually* based on these characteristics such that applications that are vulnerable to interference are ranked higher and iii) *prioritize* requests based on the computed ranking.

We observe that there are two major problems with past ranking-based schedulers. First, such schedulers trade off hardware complexity in order to improve performance or fairness. They incur high hardware complexity (logic and storage overhead as well as critical path latency) to schedule requests based on a scheme that ranks individual applications with a total order. As a result, the critical path latency and chip area cost of such schedulers are significantly higher compared to application-unaware schedulers. For example, as we demonstrate in Section 7.2, based on our RTL designs, TCM [7], a state-of-the-art application-aware scheduler is $8\times$ slower and $1.8\times$ larger than a commonly-employed application-unaware scheduler, FRFCFS [17]. Second, such schedulers not only increase hardware complexity, but also cause unfair slowdowns. When a total order based ranking is employed, applications that are at the bottom of the ranking stack get heavily deprioritized and unfairly slowed down. This greatly degrades system fairness.

Our goal, in this work, is to design a new memory scheduler that does not suffer from these two problems: one that achieves high system performance *and* fairness *while*

- L. Subramanian is currently with Intel Labs. She carried out this work when she was a PhD student with the Department of Electrical and Computer Engineering, Carnegie Mellon University.
  E-mail: lavanya.subramanian@intel.com.
- D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu are with the Carnegie Mellon University. E-mail: {donghyu1, visesh, harshar, onur}@cmu.edu.

incurring low hardware cost and low scheduling latency. To this end, we propose the *Blacklisting memory scheduler (BLISS)*. Our BLISS design is based on two new observations.

Observation 1. In contrast to forming a total rank order of all applications, we find that, to mitigate interference, it is sufficient to i) separate applications into *only two* groups, one group containing applications that are vulnerable to interference and another containing applications that cause interference, and ii) prioritize the requests of the *vulnerable-to-interference* group over the requests of the *interference-causing* group. Our approach overcomes the two major problems with such schedulers that employ per-application ranking. First, separating applications into only two groups, as opposed to employing ranking based on a total order of applications, significantly reduces hardware complexity (Section 7.2). Second, since our approach prioritizes only one dynamically-determined group of applications over another dynamically-determined group, no single application is heavily deprioritized, improving overall system fairness (Section 7).

Observation 2. We observe that applications can be efficiently classified as either *vulnerable-to-interference* or *interference-causing* by simply counting the number of consecutive requests served from an application in a short time interval. Applications with a large number of consecutively-served requests are classified as interference-causing. The rationale behind this approach is that when a large number of consecutive requests are served from the same application, requests of other applications are more likely to be delayed, causing those applications to stall. On the other hand, applications with very few consecutive requests will likely not delay other applications and are, in fact, vulnerable to interference from other applications that have a large number of requests generated and served. Our approach to classifying applications is simpler to implement than prior approaches (e.g., [4], [6], [7]) that use more complicated metrics such as memory intensity, row-buffer locality, bank-level parallelism or long-term memory service as proxies for vulnerability to interference (Section 7.2).

Mechanism overview. Based on these two observations, our mechanism, the Blacklisting Memory Scheduler, counts the number of consecutive requests served from the same application within a short time interval. When this count exceeds a threshold, BLISS places the application in the interference-causing group, which we also call the *blacklisted* group. In other words, BLISS *blacklists* the application such that it is deprioritized. During scheduling, non-blacklisted (vulnerable-to-interference) applications' requests are given higher priority over requests of blacklisted (interference-causing) applications. No per-application ranking is employed. Prioritization is based solely on two groups as opposed to a total order of applications.

This paper makes the following contributions:

- We present two new observations on how a simple grouping scheme that avoids per-application ranking can mitigate interference, based on our analyses and studies of previous memory schedulers. These observations can enable simple and effective memory interference mitigation techniques including and beyond the ones we propose in this work.

- We propose the Blacklisting memory scheduler, which achieves high system performance and fairness while incurring low hardware cost and complexity.

- We provide a comprehensive complexity analysis of five previously proposed memory schedulers, comparing their critical path latency and area via RTL implementations (Section 7.2). Our results show that BLISS reduces critical path latency/area of the memory scheduler by 79%/43% respectively, compared to the best-performing ranking-based scheduler, TCM [7].

- We evaluate BLISS against five previously-proposed memory schedulers in terms of system performance and fairness across a wide range of workloads (Section 7). Our results show that BLISS achieves 5 percent better system performance and 25 percent better fairness than the best-performing previous scheduler, TCM [7].

- We evaluate the trade-off space between performance, fairness and hardware complexity for five previously-proposed memory schedulers and BLISS (Section 7.3). We demonstrate that BLISS achieves the best trade-off between performance, fairness and complexity, compared to previous memory schedulers.

2 BACKGROUND AND MOTIVATION

In this section, we first provide a brief background on the organization of a DRAM main memory system. We then describe previous memory scheduling proposals and their shortcomings that motivate the need for a new memory scheduler - our blacklisting memory scheduler.

2.1 DRAM Background

The DRAM main memory system is organized hierarchically as channels, ranks and banks. Channels are independent and can operate in parallel. Each channel consists of ranks (typically 1 - 4) that share the command, address and data buses of the channel. A rank consists of multiple banks that can operate in parallel. However, all banks within a channel share the command, address and data buses of the channel. Each bank is organized as a two-dimensional array of rows and columns. On a data access, the entire row containing the data is brought into an internal structure called the row buffer. Therefore, a subsequent access to the same row can be served from the row buffer itself and need not access the array. Such an access is called a row hit. On an access to a different row, however, the array itself needs to be accessed. Such an access is called a row miss/conflict. A row hit is served ~2-3x faster than a row miss/conflict [18]. For more detail on DRAM operation, we refer the reader to [19], [20], [21], [22].

2.2 Memory Scheduling

Commonly employed memory controllers employ a memory scheduling policy called First Ready First Come First Served (FRFCFS) [17], [23] that leverages the row buffer by prioritizing row hits over row misses/conflicts. Older requests are then prioritized over newer requests. FRFCFS aims to maximize DRAM throughput by prioritizing row

hits. However, it unfairly prioritizes requests of applications that generate a large number of requests to the same row (high-row-buffer-locality) and access memory frequently (high-memory-intensity) [2], [3]. Previous work (e.g., [3], [4], [5], [6], [7]) proposed application-aware memory scheduling techniques that take into account the memory access characteristics of applications and schedule requests appropriately in order to mitigate inter-application interference and improve system performance and fairness. We will focus on four state-of-the-art schedulers, which we evaluate quantitatively in Section 7.

Mutlu and Moscibroda propose PARBS [4], an application-aware memory scheduler that batches the oldest requests from applications and prioritizes the batched requests, with the goals of preventing starvation and improving fairness. Within each batch, PARBS ranks individual applications based on the number of outstanding requests of each application and, using this total rank order, prioritizes requests of applications that have low-memory-intensity to improve system throughput.

Kim et al. [6] observe that applications that receive low memory service tend to experience interference from applications that receive high memory service. Based on this observation, they propose ATLAS, an application-aware memory scheduling policy that ranks individual applications based on the amount of long-term memory service each receives and prioritizes applications that receive low memory service, with the goal of improving overall system throughput.

Thread cluster memory scheduling (TCM) [7] ranks individual applications by memory intensity such that low-memory-intensity applications are prioritized over high-memory-intensity applications (to improve system throughput). Kim et al. [7] also observed that ranking all applications based on memory intensity and prioritizing low-memory-intensity applications could slow down the deprioritized high-memory-intensity applications significantly and unfairly. With the goal of mitigating this unfairness, TCM clusters applications into low and high memory-intensity clusters and employs a different ranking scheme in each cluster. In the low-memory-intensity cluster, applications are ranked by memory intensity, whereas, in the high-memory-intensity cluster, applications' ranks are shuffled to provide fairness. Both clusters employ a total rank order among applications at any given time.

More recently, Ghose et al. [8] propose a memory scheduler that aims to prioritize *critical* memory requests that stall the instruction window for long lengths of time. The scheduler predicts the criticality of a load instruction based on how long it has stalled the instruction window in the past (using the instruction address (PC)) and prioritizes requests from load instructions that have large total and maximum stall times measured over a period of time. Although this scheduler is not application-aware, we compare to it as it is the most recent scheduler that aims to maximize performance by mitigating memory interference.

2.3 Shortcomings of Previous Schedulers

These state-of-the-art schedulers attempt to achieve two main goals - high system performance and high fairness. However, previous schedulers have two major shortcomings. First, these schedulers increase hardware complexity in
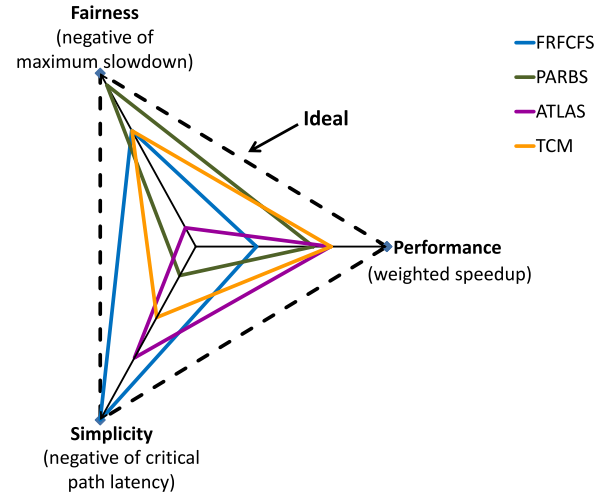

Fig. 1. Performance versus fairness versus simplicity.

order to achieve high system performance and fairness. Specifically, most of these schedulers rank individual applications with a total order, based on their memory access characteristics (e.g., [4], [5], [6], [7]). Scheduling requests based on a total rank order incurs high hardware complexity, as we demonstrate in Section 7.2, slowing down the memory scheduler significantly (by 8× for TCM compared to FRFCFS), while also increasing its area (by 1.8×). Such high critical path delays in the scheduler directly increase the time it takes to schedule a request, potentially making the memory controller latency a bottleneck. Second, a total-order ranking is unfair to applications at the bottom of the ranking stack. Even shuffling the ranks periodically (like TCM does) does not fully mitigate the unfairness and slowdowns experienced by an application when it is at the bottom of the ranking stack, as we show in Section 3.

Fig. 1 compares four major previous schedulers using a three-dimensional plot with performance, fairness and simplicity on three different axes.[1] On the fairness axis, we plot the negative of maximum slowdown, and on the simplicity axis, we plot the negative of critical path latency. Hence, the ideal scheduler would have high performance, fairness and simplicity, as indicated by the black triangle. As can be seen, previous ranking-based schedulers, PARBS, ATLAS and TCM, increase complexity significantly, compared to the currently employed FRFCFS scheduler, in order to achieve high performance and/or fairness.

Our goal, in this work, is to design a new memory scheduler that does not suffer from these shortcomings: one that achieves high system performance and fairness *while* incurring low hardware cost and complexity. To this end, we propose the *Blacklisting memory scheduler* based on two new observations described in the next section.

3 KEY OBSERVATIONS

As we described in the previous section, several major state-of-the-art memory schedulers rank individual applications with a total order, to mitigate inter-application interference. While such ranking is one way to mitigate interference, it has shortcomings, as described in Section 2.3. We seek to

---

1. Results across 80 simulated workloads on a 24-core, 4-channel system. Section 6 describes our methodology and metrics.
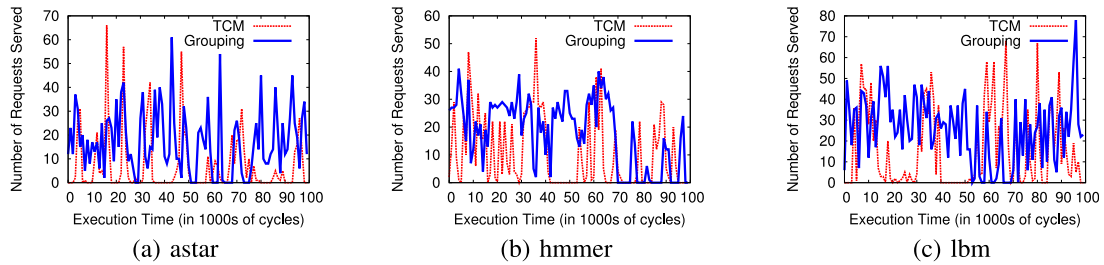
Fig. 2. Request service distribution over time with TCM and Grouping schedulers.

overcome these shortcomings by exploring an alternative means to protecting vulnerable applications from interference. We make two key observations on which we build our new memory scheduling mechanism.

Observation 1. Separating applications into only two groups (interference-causing and vulnerable-to-interference), without ranking individual applications using a total order, is sufficient to mitigate inter-application interference. This leads to higher performance, fairness and lower complexity, all at the same time.

We observe that applications that are vulnerable to interference can be protected from interference-causing applications by simply separating them into two groups, one containing interference-causing applications and another containing vulnerable-to-interference applications, rather than ranking individual applications with a total order as many state-of-the-art schedulers do. To motivate this, we contrast TCM [7], which clusters applications into two groups and employs a total rank order within each cluster, with a simple scheduling mechanism (*Grouping*) that simply groups applications only into two groups, based on memory intensity (as TCM does), and prioritizes the low-intensity group *without* employing ranking in each group. *Grouping* uses the FRFCFS policy within each group. Fig. 2 shows the number of requests served during a 100,000 cycle period at intervals of 1,000 cycles, for three representative applications, astar, hmmer and lbm from the SPEC CPU2006 benchmark suite [24], using these two schedulers. All these three applications are in the high-memory-intensity group.[2] These three applications are executed with other applications in a simulated 24-core 4-channel system.[3]

Fig. 2 shows that TCM has high variance in the number of requests served across time, with very few requests being served during several intervals and many requests being served during a few intervals. This behavior is seen in most applications in the high-memory-intensity cluster since TCM ranks individual applications with a total order. This ranking causes some high-memory-intensity applications' requests to be prioritized over *other* high-memory-intensity applications' requests, at any point in time, resulting in high interference. Although TCM periodically shuffles this total-order ranking, we observe that an application benefits from ranking *only* during those periods when it is ranked very high. These very highly ranked periods correspond to the spikes in the number of requests served (for TCM) in Fig. 2

for that application. During the other periods of time when an application is ranked lower (i.e., most of the *shuffling intervals*), only a small number of its requests are served, resulting in very slow progress. Therefore, most high-memory-intensity applications experience high slowdowns due to the total-order ranking employed by TCM.

On the other hand, when applications are separated into only two groups based on memory intensity and no per-application ranking is employed within a group, some interference exists among applications within each group (due to the application-unaware FRFCFS scheduling in each group). In the high-memory-intensity group, this interference contributes to the few low-request-service periods seen for *Grouping* in Fig. 2. However, the request service behavior of *Grouping* is less spiky than that of TCM, resulting in lower memory stall times and a more steady and overall higher progress rate for high-memory-intensity applications, as compared to when applications are ranked in a total order. In the low-memory-intensity group, there is not much of a difference between TCM and *Grouping*, since applications anyway have low memory intensities and hence, do not cause significant interference to each other. Therefore, *Grouping* results in higher system performance and significantly higher fairness than TCM, as shown in Fig. 3 (across 80 24-core workloads on a simulated 4-channel system).

Solely grouping applications into two also requires much lower hardware overhead than ranking-based schedulers that incur high overhead for computing and enforcing a total rank order for all applications. Therefore, grouping can not only achieve better system performance and fairness than ranking, but it also can do so while incurring lower hardware cost. However, classifying applications into two groups at coarse time granularities, on the order of a few million cycles, like TCM's clustering mechanism does (and like what we have evaluated in Fig. 3), can still cause unfair application slowdowns. This is because applications in one group would be deprioritized for a long time interval, which is especially dangerous if application behavior changes during the interval. Our second observation, which we describe next, minimizes such unfairness and at the same time reduces the complexity of grouping even further.
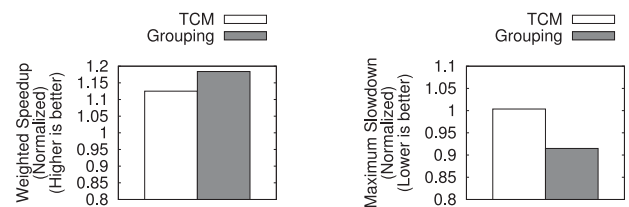
---

2. We found very similar behavior in all other such applications we examined.

3. See Section 6 for our methodology.

Fig. 3. Performance and fairness of Grouping versus TCM.

Observation 2. Applications can be classified into interference-causing and *vulnerable-to-interference* groups by monitoring the number of consecutive requests served from each application at the memory controller. This leads to higher fairness and lower complexity than grouping schemes that rely on coarse-grained memory intensity measurement.

Previous work actually attempted to perform grouping, along with ranking, to mitigate interference. As described in Section 2, TCM [7] ranks applications by memory intensity and classifies applications that make up a certain fraction of the total memory bandwidth usage into a *group* called the *low-memory-intensity cluster* and the remaining applications into a second group called the *high-memory-intensity cluster*. While employing such a grouping scheme, without ranking individual applications, reduces hardware complexity and unfairness compared to a total order based ranking scheme (as we show in Fig. 3), it i) *can still cause unfair slowdowns due to classifying applications into groups at coarse time granularities*, and ii) *incurs additional hardware overhead and scheduling latency to compute and rank applications by long-term memory intensity and memory bandwidth usage*.

We propose to perform application grouping using a significantly simpler, novel scheme: simply by counting the number of requests served from each application in a short time interval. Applications that have a large number (i.e., above a threshold value) of consecutive requests served are classified as interference-causing (this classification is periodically reset). The rationale behind this scheme is that when an application has a large number of consecutive requests served within a short time period, which is typical of applications with high memory intensity or high row-buffer locality, it delays other applications' requests, thereby stalling their progress. Hence, identifying and essentially *blacklisting* such interference-causing applications by placing them in a separate group and deprioritizing requests of this blacklisted group can prevent such applications from hogging the memory bandwidth. As a result, the interference experienced by vulnerable applications is mitigated. The blacklisting classification is cleared periodically, at short time intervals (on the order of 1,000 s of cycles) in order not to deprioritize an application for too long of a time period to cause unfairness or starvation. Such clearing and re-evaluation of application classification at short time intervals significantly reduces unfair application slowdowns (as we quantitatively show in Section 7.7), while reducing complexity compared to tracking per-application metrics such as memory intensity.

4 MECHANISM

In this section, we present the details of our Blacklisting memory scheduler that employs a simple grouping scheme motivated by our key observations from Section 3. The two main components of BLISS are i) the blacklisting mechanism that separates applications into the blacklisted (interference-causing) and non-blacklisted (vulnerable-to-interference) groups and ii) the memory scheduling mechanism that schedules requests based on the blacklisting mechanism. We describe each in turn.

4.1 The Blacklisting Mechanism

The blacklisting mechanism needs to keep track of three quantities: 1) the application (i.e., hardware context) ID of the last scheduled request (*Application ID*),[4] 2) the number of requests served from an application (*#Requests Served*), and 3) the blacklist status of each application.

When the memory controller is about to issue a request, it compares the application ID of the request with the *Application ID* of the *last scheduled request*.

- If the application IDs of the two requests are the same, the *#Requests Served* counter is incremented.
- If the application IDs of the two requests are not the same, the *#Requests Served* counter is reset to zero and the *Application ID* register is updated with the application ID of the request that is being issued.

If the *#Requests Served* exceeds a *Blacklisting Threshold* (set to 4 in most of our evaluations):

- The application with ID *Application ID* is blacklisted (classified as interference-causing).
- The *#Requests Served* counter is reset to zero.

The blacklist information is cleared periodically after every *Clearing Interval* (10,000 cycles in our major evaluations).

4.2 Blacklist-Based Memory Scheduling

Once the blacklist information is computed, it is used to determine the scheduling priority of a request. Memory requests are prioritized in the following order:

1) Non-blacklisted applications' requests
2) Row-buffer hit requests
3) Older requests

Prioritizing requests of non-blacklisted applications over requests of blacklisted applications mitigates interference. Row-buffer hits are then prioritized to optimize DRAM bandwidth utilization. Finally, older requests are prioritized over younger requests for forward progress.

5 IMPLEMENTATION

The Blacklisting memory scheduler requires additional storage (flip flops) and logic over an FRFCFS scheduler to 1) perform blacklisting and 2) prioritize non-blacklisted applications' requests. We analyze the storage and logic cost of it.

5.1 Storage Cost

In order to perform blacklisting, the memory scheduler needs the following storage components:

- one register to store *Application ID*
- one counter for *#Requests Served*
- one register to store the *Blacklisting Threshold* that determines when an application should be blacklisted
- a blacklist bit vector to indicate the blacklist status of each application (one bit for each hardware context)

---

4. An application here denotes a hardware context. There can be as many applications executing actively as there are hardware contexts. Multiple hardware contexts belonging to the same application are considered separate applications by our mechanism, but our mechanism can be extended to deal with such multithreaded applications.

TABLE 1
Configuration of the Simulated System

Processor	16-64 cores, 5.3 GHz, 3-wide issue, 128-entry instruction window
Last-level cache	64B cache-line, 16-way associative, 512 KB private cache-slice per core
Memory controller	128-entry read/write request queue per controller
Memory	Timing: DDR3-1066 (8-8-8) [29] Organization: 1-8 channels, 1 rank-per-channel, 8 banks-per-rank, 8 KB row-buffer

In order to prioritize non-blacklisted applications' requests, the memory controller needs to store the application (hardware context) ID of each request so it can determine the blacklist status of the application and appropriately schedule the request.

5.2 Logic Cost

The memory scheduler requires comparison logic to

- determine when an application's *#Requests Served* exceeds the *Blacklisting Threshold* and set the bit corresponding to the application in the *Blacklist* bit vector.
- prioritize non-blacklisted applications' requests.

We provide a detailed quantitative evaluation of the hardware area cost and logic latency of implementing BLISS and previously proposed memory schedulers, in Section 7.2.

6 METHODOLOGY

6.1 System Configuration

We model the DRAM memory system using a cycle-level in-house DDR3-SDRAM simulator. The DRAM model we use is similar to Ramulator's DDR3 model [25], which is publicly available [26]. This DDR3 simulator is integrated with a cycle-level in-house simulator that models out-of-order execution cores, driven by a Pin [27] tool at the frontend. We have made the simulator publicly available at [28]. The core model includes an instruction window that keeps track of ordering and retiring instructions in the right order. The execution of Load/Store instructions (through the caches/main memory) is modeled in detail, while single cycle execution is assumed for other non-memory instructions. Each core has a private cache of 512 KB size, which connects directly to the memory controller. We present most of our results on a system with the DRAM main memory as the only shared resource in order to isolate the effects of memory bandwidth interference on application performance. We also present results with shared caches in Section 7.11. We model contention between cores at the cache and main memory. Table 1 provides more details of our simulated system. We perform most of our studies on a system with 24 cores and four channels. We provide a sensitivity analysis for a wide range of core and channel counts, in Section 7.11. Each channel has one rank and each rank has eight banks. We stripe data across channels and banks at the granularity of a row.

6.2 Workloads

We perform our main studies using 24-core multiprogrammed workloads made of applications from the SPEC CPU2006 suite [24], TPC-C, Matlab and the NAS parallel benchmark suite [30].[5] We classify a benchmark as memory-intensive if it has a Misses Per Kilo Instruction (MPKI) greater than 5 and memory-non-intensive otherwise. We construct four categories of workloads (with 20 workloads in each category), with 25, 50, 75 and 100 percent of memory-intensive applications. This makes up a total of 80 workloads with a range of memory intensities, constructed using random combinations of benchmarks, modeling a cloud computing like scenario where workloads of various types are consolidated on the same node to improve efficiency. We also evaluate 16-, 32- and 64- core workloads, with different memory intensities, created using a similar methodology as described above for the 24-core workloads. We simulate each workload for 100 million representative cycles, as done by previous studies in memory scheduling [4], [6], [7].

6.3 Metrics

We quantitatively compare BLISS with previous memory schedulers in terms of system performance, fairness and complexity. We use the weighted speedup [31], [32], [33] metric to measure system performance. We use the maximum slowdown metric [6], [7], [31], [34] to measure unfairness. We report the harmonic speedup metric [35] as another measure of system performance. The harmonic speedup metric also serves as a measure of balance between system performance and fairness [35]. We report area in $micrometer^2$ (um^2) and scheduler critical path latency in nanoseconds (ns) as measures of complexity.

6.4 RTL Synthesis Methodology

In order to obtain timing/area results for BLISS and previous schedulers, we implement them in Register Transfer Level (RTL), using Verilog. We synthesize the RTL implementations with a commercial 32 nm standard cell library, using the Design Compiler tool from Synopsys.

6.5 Mechanism Parameters

For BLISS, we use a value of four for *Blacklisting Threshold*, and a value of 10,000 cycles for *Clearing Interval*. These values provide a good balance between performance and fairness, as we observe from our sensitivity studies in Section 7.12. For the other schedulers, we tuned their parameters to achieve high performance and fairness on our system configurations and workloads. We use a *Marking-Cap* of 5 for PARBS, cap of 4 for FRFCFS-Cap, *HistoryWeight* of 0.875 for ATLAS, *ClusterThresh* of 0.2 and *ShuffleInterval* of 1,000 cycles for TCM.

7 EVALUATION

We compare BLISS with five previously proposed memory schedulers, FRFCFS, FRFCFS with a cap (FRFCFS-Cap) [3], PARBS, ATLAS and TCM. FRFCFS-Cap is a modified version of FRFCFS that caps the number of consecutive row-buffer hitting requests that can be served from an application [3]. Fig. 4 shows the average system performance (weighted speedup and harmonic speedup) and unfairness (maximum slowdown) across all our workloads. Fig. 5 shows a Pareto

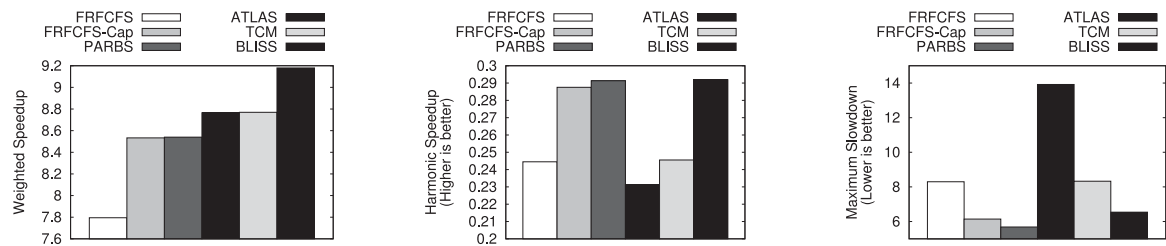---

5. Each benchmark is single threaded.

Fig. 4. System performance and fairness of BLISS compared to previous schedulers.

plot of weighted speedup and maximum slowdown. We make three major observations. First, BLISS achieves 5 percent better weighted speedup, 25 percent lower maximum slowdown and 19 percent better harmonic speedup than the best performing previous scheduler (in terms of weighted speedup), TCM, while reducing the critical path and area by 79 and 43 percent respectively (as we will show in Section 7.2). Therefore, we conclude that BLISS achieves both high system performance and fairness, at low hardware cost and complexity.

Second, BLISS significantly outperforms all these five previous schedulers in terms of system performance, however, it has 10 percent higher unfairness than PARBS, the previous scheduler with the least unfairness. PARBS creates request batches containing the oldest requests from each application. Older batches are prioritized over newer batches. However, within each batch, individual applications' requests are ranked and prioritized based on memory intensity. PARBS aims to preserve fairness by batching older requests, while still employing ranking within a batch to prioritize low-memory-intensity applications. We observe that the batching aspect of PARBS is quite effective in mitigating unfairness, although it increases complexity. This unfairness reduction also contributes to the high harmonic speedup of PARBS. However, batching restricts the amount of request reordering that can be achieved through ranking. Hence, low-memory-intensity applications that would benefit from prioritization via aggressive request reordering have lower performance. As a result, PARBS has 8 percent lower weighted speedup than BLISS. Furthermore, PARBS has a $6.5\times$ longer critical path and $\sim 2\times$ greater area than BLISS, as we will show in Section 7.2. Therefore, we conclude that BLISS achieves better system performance than PARBS, at much lower hardware cost, while slightly trading off fairness.

Third, BLISS has 4 percent higher unfairness than FRFCFS-Cap, but it also 8 percent higher performance than

FRFCFS-Cap. FRFCFS-Cap has higher fairness than BLISS since it restricts the length of *only* the *ongoing* row hit streak, whereas blacklisting an application can deprioritize the application *for a longer time*, until the next clearing interval. As a result, FRFCFS-Cap slows down high-row-buffer-locality applications to a lower degree than BLISS. However, restricting *only* the on-going streak rather than blacklisting an interfering application for a longer time causes more interference to other applications, degrading system performance compared to BLISS. Furthermore, FRFCFS-Cap is unable to mitigate interference due to applications with high memory intensity yet low-row-buffer-locality, whereas BLISS is effective in mitigating interference due to such applications as well. Hence, we conclude that BLISS achieves higher performance (weighted speedup) than FRFCFS-Cap, while slightly trading off fairness.

7.1 Analysis of Individual Workloads

In this section, we analyze the performance and fairness for individual workloads, when employing different schedulers. Fig. 6 shows the performance and fairness normalized
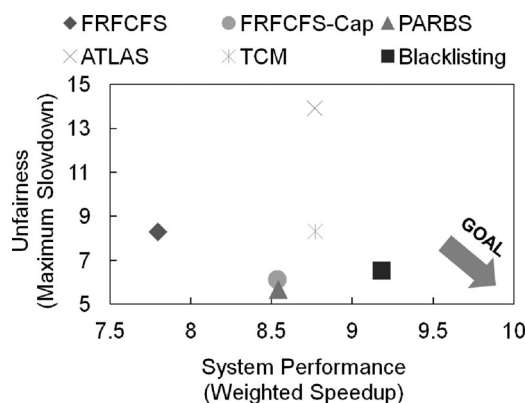

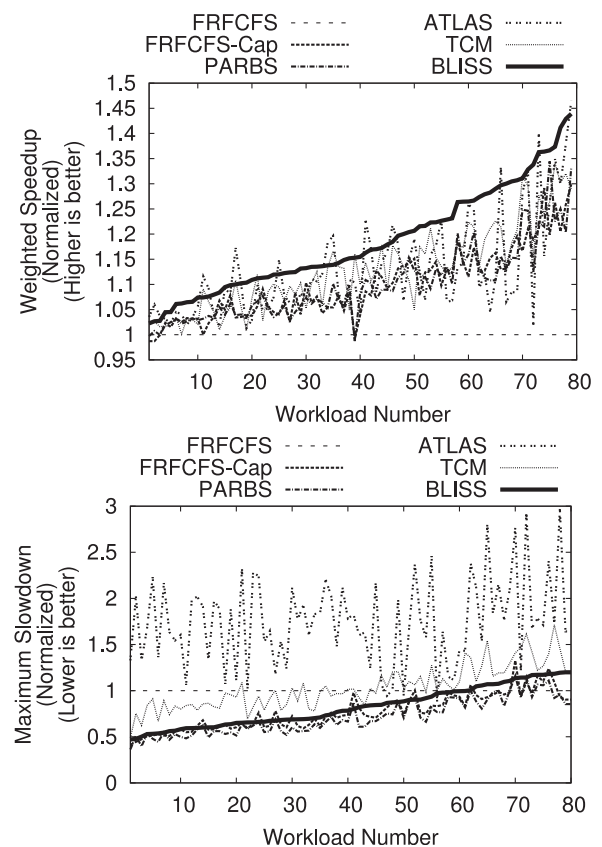Fig. 5. Pareto plot of system performance and fairness.

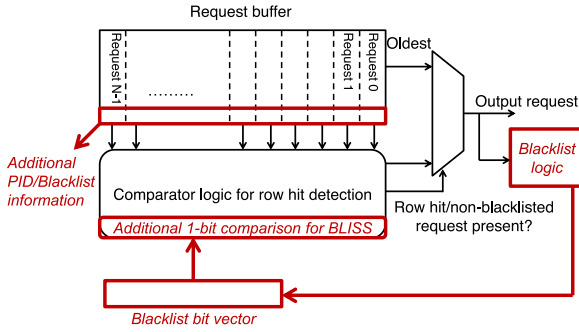Fig. 6. System performance and fairness for all workloads.
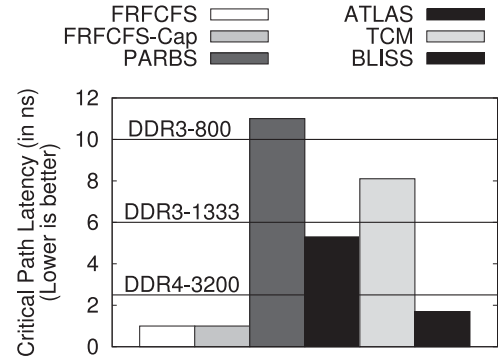
Fig. 7. Block diagram for FRFCFS with extensions for BLISS.

Fig. 8. Critical path: BLISS versus previous schedulers.

Fig. 9. Area: BLISS versus previous schedulers.

to the baseline FRFCFS scheduler for all our 80 workloads, for BLISS and previous schedulers, in the form of S-curves [36]. The workloads are sorted based on the performance improvement of BLISS. We draw three major observations. First, BLISS achieves the best performance among all previous schedulers for most of our workloads. For a few workloads, ATLAS achieves higher performance, by virtue of always prioritizing applications that receive low memory service. However, always prioritizing applications that receive low memory service can unfairly slow down applications with high memory intensities, thereby degrading fairness significantly (as shown in the maximum slowdown plot, Fig. 6 bottom). Second, BLISS achieves significantly higher fairness than ATLAS and TCM, the best-performing previous schedulers, while also achieving higher performance than them and approaches the fairness of the fairest previous schedulers, PARBS and FRFCFS-Cap. As described in the analysis of average performance and fairness results above, PARBS, by virtue of request batching and FRFCFS-Cap, by virtue of restricting only the current row hit streak achieve higher fairness (lower maximum slowdown) than BLISS for a number of workloads. However, these schedulers achieve higher fairness at the cost of lower system performance, as shown in Fig. 6. Third, for some workloads with very high memory intensities, the default FRFCFS scheduler achieves the best fairness. This is because memory bandwidth becomes a very scarce resource when the memory intensity of a workload is very high. Hence, prioritizing row hits utilizes memory bandwidth efficiently for such workloads, thereby resulting in higher fairness. Based on these observations, we conclude that BLISS achieves the best performance and a good trade-off between fairness and performance for most of the workloads we examine.

7.2 Hardware Complexity

Figs. 8 and 9 show the critical path latency and area of five previous schedulers and BLISS for a 24-core system for every memory channel. We draw two major conclusions. First, previously proposed ranking-based schedulers, PARBS/ATLAS/TCM, greatly increase the critical path latency and area of the memory scheduler: by $11\times/5.3\times/8.1\times$ and $2.4\times/1.7\times/1.8\times$ respectively, compared to FRFCFS and FRFCFS-Cap, whereas BLISS increases latency and area by only $1.7\times$ and 3.2 percent over FRFCFS/FRFCFS-Cap.[6] Fig. 7 shows a high-level block diagram for FRFCFS and the changes
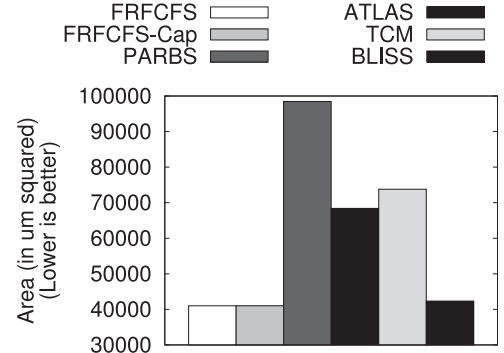
required for BLISS over the FRFCFS scheduler (in red and italics). Second, PARBS, ATLAS and TCM cannot meet the stringent worst-case timing requirements posed by the DDR3 and DDR4 standards [18], [37]. In the case where every request is a row-buffer hit, the memory controller would have to schedule a request every read-to-read cycle time (t_{CCD}), the minimum value of which is 4 cycles for both DDR3 and DDR4. TCM and ATLAS can meet this worst-case timing only until DDR3-800 (read-to-read cycle time of 10 ns) and DDR3-1333 (read-to-read cycle time of 6 ns) respectively, whereas BLISS can meet the worst-case timing all the way down to DDR4-3200 (read-to-read time of 2.5 ns). Hence, the high critical path latency of PARBS, ATLAS and TCM is a serious impediment to their adoption in today's and future memory interfaces. Techniques like pipelining could potentially be employed to reduce the critical path latency of these previous schedulers. However, the additional flops required for pipelining would increase area, power and design effort significantly. Therefore, we conclude that BLISS, with its greatly lower complexity and cost as well as higher system performance and competitive or better fairness, is a more effective alternative to state-of-the-art application-aware memory schedulers.

7.3 Analysis of Trade-offs Between Performance, Fairness and Complexity

In the previous sections, we studied the performance, fairness and complexity of different schedulers individually. In this section, we will analyze the trade-offs between these metrics for different schedulers. Fig. 10 shows a three-dimensional radar plot with performance, fairness and simplicity on three different axes. On the fairness axis, we plot the negative of the maximum slowdown numbers, and on the simplicity axis, we

---

6. The area numbers are for the lowest value of critical path latency that the scheduler is able to meet.
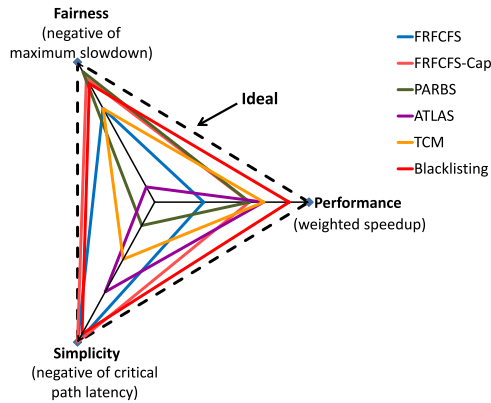
Fig. 10. Performance, fairness and simplicity trade-offs.

plot the negative of the critical path latency numbers. Hence, the ideal scheduler would have high performance, fairness and simplicity, as indicated by the encompassing, dashed black triangle. We draw three major conclusions about the different schedulers we study. First, application-unaware schedulers, such as FRFCFS and FRFCFS-Cap, are simple. However, they have low performance and/or fairness. Second, application-aware schedulers, such as PARBS, ATLAS and TCM, improve performance or fairness by ranking based on applications' memory access characteristics. However, they do so at the cost of increasing complexity (reducing simplicity) significantly. Third, BLISS, achieves high performance and fairness, while keeping the design simple, thereby approaching the ideal scheduler design (i.e., leading to a triangle that is closer to the ideal triangle). This is because BLISS requires only simple hardware changes to the memory controller to blacklist applications that have long streaks of requests served, which effectively mitigates interference. Therefore, we conclude that BLISS achieves the best trade-off between performance, fairness and simplicity.

7.4 Understanding the Benefits of BLISS

We present the distribution of the number of consecutive requests served (streaks) from individual applications to better understand why BLISS effectively mitigates interference. Fig. 11 shows the distribution of requests served across different streak lengths ranging from 1 to 16 for FRFCFS, PARBS, TCM and BLISS for *six representative applications from the same 24-core workload.*[7] The figure captions indicate the memory intensity, in misses per kilo instruction (MPKI) and row-buffer hit rate (RBH) of each application when it is run alone. Figs. 11a, 11b and 11c show the streak length distributions of applications that have a tendency to cause interference (*libquantum, mcf* and *lbm*). All these applications have high memory intensity and/or high row-buffer locality. Figs. 11d, 11e and 11f show applications that are vulnerable to interference (*calculix, cactusADM* and *sphinx3*). These applications have lower memory intensities and row-buffer localities, compared to the interference-causing applications. We observe that BLISS shifts the distribution of streak lengths towards the left for the interference-causing applications, while it shifts the streak length distribution to the right for the interference-prone applications. Hence, BLISS breaks long streaks of consecutive requests for

interference-causing applications, while enabling longer streaks for vulnerable applications. This enables such vulnerable applications to make faster progress, thereby resulting in better system performance and fairness. We have observed similar results for most of our workloads.

7.5 Average Request Latency

In this section, we evaluate the average memory request latency (from when a request is generated until when it is served) metric and seek to understand its correlation with performance and fairness. Fig. 12 presents the average memory request latency (from when the request is generated until when it is served) for the five previously proposed memory schedulers and BLISS. Two major observations are in order. First, FRFCFS has the lowest average request latency among all the schedulers. This is expected since FRFCFS maximizes DRAM throughput by prioritizing row-buffer hits. Hence, the number of requests served is maximized overall (across all applications). However, maximizing throughput (i.e., minimizing overall average request latency) degrades the performance of low-memory-intensity applications, since these applications' requests are often delayed behind row-buffer hits and older requests. This results in degradation in system performance and fairness, as shown in Fig. 4.

Second, ATLAS and TCM, memory schedulers that prioritize requests of low-memory-intensity applications by employing a full ordered ranking achieve relatively low average latency. This is because these schedulers reduce the latency of serving requests from latency-critical, low-memory-intensity applications significantly. Furthermore, prioritizing low-memory-intensity applications' requests does not increase the latency of high-memory-intensity applications significantly. This is because high-memory-intensity applications already have high memory access latencies (even when run alone) due to queueing delays. Hence, average request latency does not increase much from deprioritizing requests of such applications. However, always prioritizing such latency-critical applications results in lower memory throughput for high-memory-intensity applications, resulting in unfair slowdowns (as we show in Fig. 4). Third, memory schedulers that provide the best fairness, PARBS, FRFCFS-Cap and BLISS have high average memory latencies. This is because these schedulers, while employing techniques to prevent requests of vulnerable applications with low memory intensity and low row-buffer locality from being delayed, also avoid unfairly delaying requests of high-memory-intensity applications. As a result, they do not reduce the request service latency of low-memory-intensity applications significantly, at the cost of denying memory throughput to high-memory-intensity applications, unlike ATLAS or TCM. Based on these observations, we conclude that while some applications benefit from low memory access latencies, other applications benefit more from higher memory throughput than lower latency. Hence, average memory latency is *not* a suitable metric to estimate system performance or fairness.

7.6 Impact of Clearing the Blacklist Asynchronously

The Blacklisting scheduler we have presented and evaluated so far clears the blacklisting information periodically (every 10,000 cycles in our evaluations so far), such that *all* applications are removed from the blacklist at the end of a

---

7. A value of 16 captures streak lengths 16 and above.

(a) libquantum (MPKI: 52; RBH: 99%) (b) mcf (MPKI: 146; RBH: 40%) (c) lbm (MPKI: 41; RBH: 89%)

(d) calculix (MPKI: 0.1; RBH: 85%) (e) sphinx3 (MPKI: 24; RBH: 91%) (f) cactusADM (MPKI: 7; RBH: 49%)
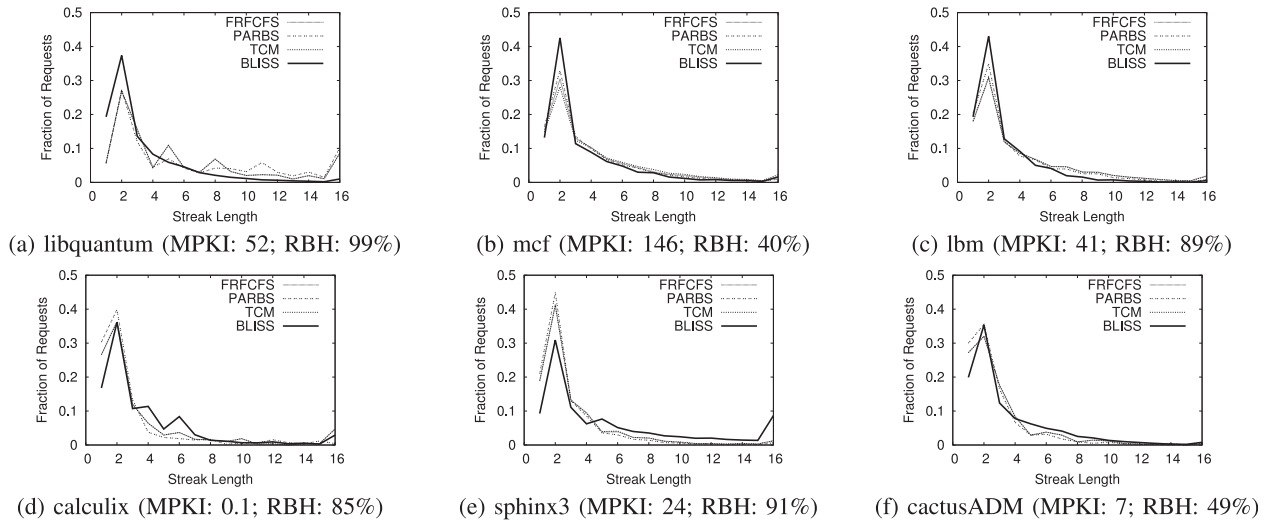
Fig. 11. Distribution of streak lengths.

Clearing Interval. In this section, we evaluate an alternative design where an individual application is removed from the blacklist *Clearing Interval* cycles after it has been blacklisted (independent of the other applications). In order to implement this alternative design, each application would need an additional counter to keep track of the number of remaining cycles until the application would be removed from the blacklist. This counter is set (to the *Clearing Interval*) when an application is blacklisted and is decremented every cycle until it becomes zero. When it becomes zero, the corresponding application is removed from the blacklist. We use a *Clearing Interval* of 10,000 cycles for this alternative design as well.

Table 2 shows the system performance and fairness of the original BLISS design (BLISS) and the alternative design in which individual applications are removed from the blacklist asynchronously (BLISS-Individual-Clearing). As can be seen, the performance and fairness of the two designs are similar. Furthermore, the first design (BLISS) is simpler since it does not need to maintain an additional counter for each application. We conclude that the original BLISS design is more efficient, in terms of performance, fairness and complexity.

7.7   Comparison with TCM's Clustering Mechanism

Fig. 13 shows the system performance and fairness of BLISS, TCM and TCM's clustering mechanism (TCM-Cluster). TCM-Cluster is a modified version of TCM that performs

clustering, but does not rank applications within each cluster. We draw two major conclusions. First, TCM-Cluster has similar system performance as BLISS, since both BLISS and TCM-Cluster prioritize vulnerable applications by separating them into a group and prioritizing that group rather than ranking individual applications. Second, TCM-Cluster has significantly higher unfairness compared to BLISS. This is because TCM-Cluster always deprioritizes high-memory-intensity applications, regardless of whether or not they are causing interference (as described in Observation 2 in Section 3). BLISS, on the other hand, observes an application at fine time granularities, independently at every memory channel and blacklists an application at a channel *only when* it is generating a number of consecutive requests (i.e., potentially causing interference to other applications).

7.8   Evaluation of Row Hit Based Blacklisting

BLISS, by virtue of restricting the number of consecutive requests that are served from an application, attempts to mitigate the interference caused by both high-memory-intensity and high-row-buffer-locality applications. In this section, we attempt to isolate the benefits from restricting consecutive row-buffer hitting requests versus non-row-buffer hitting requests. To this end, we evaluate the performance and fairness benefits of a mechanism that places an application in the blacklist when a certain number of row-buffer hitting requests (N) to the same row have been served for an application (we call this FRFCFS-Cap-Blacklisting as the scheduler essentially is FRFCFS-Cap with blacklisting). We use an N value of 4 in our evaluations.

Fig. 14 compares the system performance and fairness of BLISS with FRFCFS-Cap-Blacklisting. We make three major
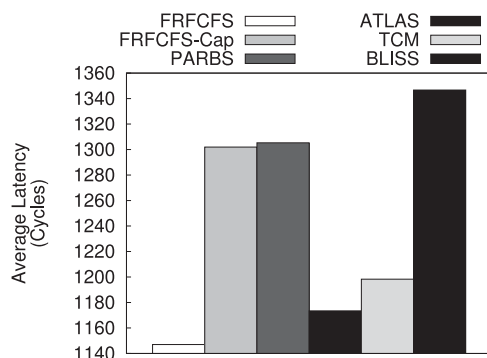

Fig. 12. The average request latency metric.

TABLE 2
Clearing the Blacklist Asynchronously

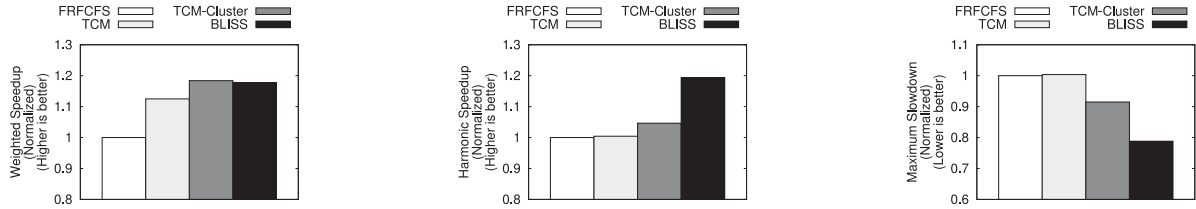| Metric	BLISS	BLISS-Individual-Clearing
Weighted Speedup	9.18	9.12
Maximum Slowdown	6.54	6.60

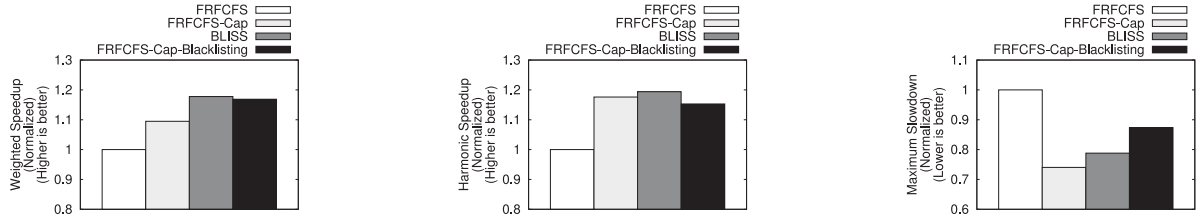Fig. 13. Comparison with TCM's clustering mechanism.

Fig. 14. Comparison with FRFCFS-Cap combined with blacklisting.

observations. First, FRFCFS-Cap-Blacklisting has similar system performance as BLISS. On further analysis of individual workloads, we find that FRFCFS-Cap-Blacklisting blacklists only applications with high row-buffer locality, causing requests of non-blacklisted high-memory-intensity applications to interfere with requests of low-memory-intensity applications. However, the performance impact of this interference is offset by the performance improvement of high-memory-intensity applications that are not blacklisted. Second, FRFCFS-Cap-Blacklisting has higher unfairness (higher maximum slowdown and lower harmonic speedup) than BLISS. This is because the high-memory-intensity applications that are not blacklisted are prioritized over the blacklisted high-row-buffer-locality applications, thereby interfering with and slowing down the high-row-buffer-locality applications significantly. Third, FRFCFS-Cap-Blacklisting requires a per-bank counter to count and cap the number of row-buffer hits, whereas BLISS needs only one counter per-channel to count the number of consecutive requests from the same application. Therefore, we conclude that BLISS is more effective in mitigating unfairness while incurring lower hardware cost, than the FRFCFS-Cap-Blacklisting scheduler that we build combining principles from FRFCFS-Cap and BLISS.

7.9 Comparison with Criticality-Aware Scheduling

We compare the system performance and fairness of BLISS with those of criticality-aware memory schedulers [8]. The basic idea behind criticality-aware memory scheduling is to prioritize memory requests from load instructions that have stalled the instruction window for long periods of time in the past. Ghose et al. [8] evaluate prioritizing load requests based on both maximum stall time (Crit-MaxStall) and total stall time (Crit-TotalStall) caused by load instructions in the

past. Fig. 15 shows the system performance and fairness of BLISS and the criticality-aware scheduling mechanisms, normalized to FRFCFS, across 40 workloads. Two observations are in order. First, BLISS significantly outperforms criticality-aware scheduling mechanisms in terms of both system performance and fairness. This is because the criticality-aware scheduling mechanisms unfairly deprioritize and slow down low-memory-intensity applications that inherently generate fewer requests, since stall times tend to be low for such applications. Second, criticality-aware scheduling incurs hardware cost to prioritize requests with higher stall times. Specifically, the number of bits to represent stall times is on the order of 12-14, as described in [8]. Hence, the logic for comparing stall times and prioritizing requests with higher stall times would incur even higher cost than per-application ranking mechanisms where the number of bits to represent a core's rank grows only as $log_2 NumberOfCores$ (e.g., 5 bits for a 32-core system). Therefore, we conclude that BLISS achieves significantly better system performance and fairness, while incurring lower hardware cost.

7.10 Effect of Workload Memory Intensity and Row-Buffer Locality

In this section, we study the impact of workload memory intensity and row-buffer locality on performance and fairness of BLISS and five previous schedulers.

Workload memory intensity. Fig. 16 shows system performance and fairness for workloads with different memory intensities, classified into different categories based on the fraction of high-memory-intensity applications in a workload.[8] We draw three major conclusions. First, BLISS outperforms previous memory schedulers in terms of system performance across all intensity categories. Second, the system performance benefits of BLISS increase with workload memory intensity. This is because as the number of high-memory-intensity applications in a workload increases, ranking individual applications, as done by previous schedulers, causes more unfairness and degrades system performance. Third, BLISS achieves significantly lower unfairness
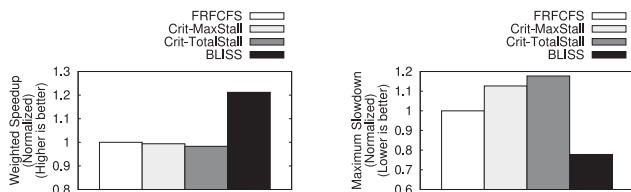

Fig. 15. Comparison with criticality-aware scheduling.

8. We classify applications with MPKI less than 5 as low-memory-intensity and the rest as high-memory-intensity.
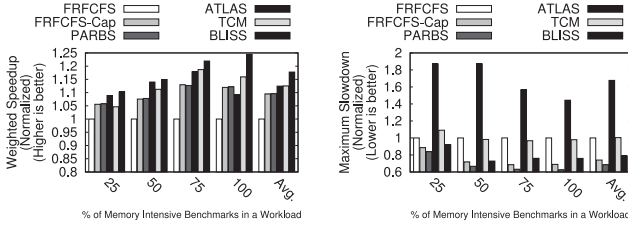
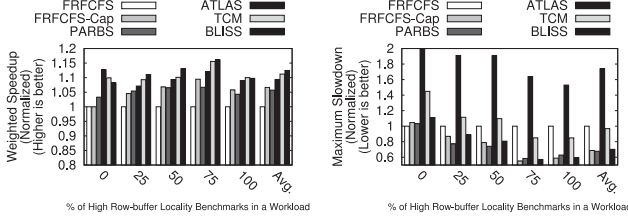Fig. 16. Sensitivity to workload memory intensity.
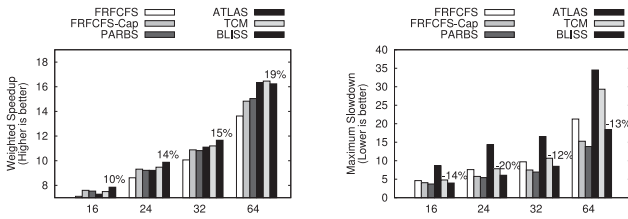

Fig. 17. Sensitivity to row-buffer locality.

Fig. 18. Sensitivity to number of cores.

than previous memory schedulers, except FRFCFS-Cap and PARBS, across all intensity categories. Therefore, we conclude that BLISS is effective in mitigating interference and improving system performance and fairness across workloads with different compositions of high- and low-memory-intensity applications.

Workload Row-Buffer Locality. Fig. 17 shows the system performance and fairness of five previous schedulers and BLISS when the number of high row-buffer locality applications in a workload is varied.[9] We draw three observations. First, BLISS achieves the best performance and close to the best fairness in most row-buffer locality categories. Second, BLISS' performance and fairness benefits over baseline FRFCFS increase as the number of high-row-buffer-locality applications in a workload increases. As the number of high-row-buffer-locality applications in a workload increases, there is more interference to the low-row-buffer-locality applications that are vulnerable. Hence, there is more opportunity for BLISS to mitigate this interference and improve performance and fairness. Third, when all applications in a workload have high row-buffer locality (100 percent), the performance and fairness improvements of BLISS over baseline FRFCFS are a bit lower than the other categories. This is because, when all applications have high row-buffer locality, they each hog the row-buffer in turn and are not as susceptible to interference as the other categories in which there are vulnerable low-row-buffer-locality applications. However, the performance/fairness benefits of BLISS are still significant since BLISS is effective in
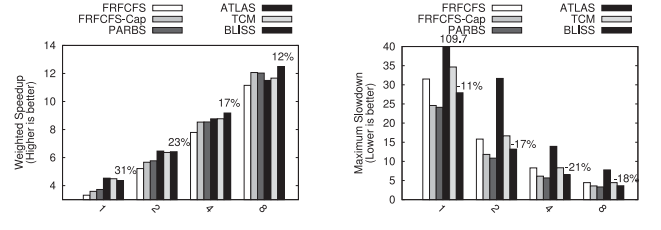

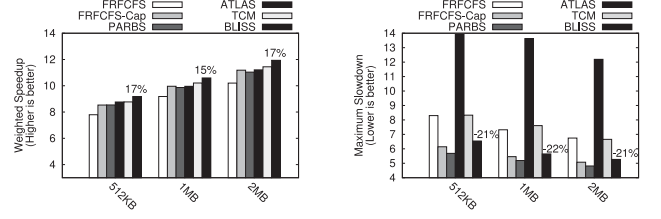
Fig. 19. Sensitivity to number of channels.

Fig. 20. Sensitivity to cache size.

regulating how the row-buffer is shared among different applications. Overall, we conclude that BLISS is effective in achieving high performance and fairness across workloads with different compositions of high- and low-row-buffer-locality applications.

7.11 Sensitivity to System Parameters

Core and channel count. Figs. 18 and 19 show the system performance and fairness of FRFCFS, PARBS, TCM and BLISS for different core counts (when the channel count is 4) and different channel counts (when the core count is 24), across 40 workloads for each core/channel count. The numbers over the bars indicate percentage increase or decrease compared to FRFCFS. We did not optimize the parameters of different schedulers for each configuration as this requires months of simulation time. We draw three major conclusions. First, the absolute values of weighted speedup increases with increasing core/channel count, whereas the absolute values of maximum slowdown increase/decrease with increasing core/channel count respectively, as expected. Second, BLISS achieves higher system performance and lower unfairness than all the other scheduling policies (except PARBS and FRFCFS-Cap, in terms of fairness) similar to our results on the 24-core, 4-channel system, by virtue of its effective interference mitigation. The only anomaly is that TCM has marginally higher weighted speedup than BLISS for the 64-core system. However, this increase comes at the cost of significant increase in unfairness. Third, BLISS' system performance benefit (as indicated by the percentages on top of the bars) increases when the system becomes more bandwidth constrained, i.e., high core counts and low channel counts. As contention increases in the system, BLISS has greater opportunity to mitigate it.[10]

Cache size. Fig. 20 shows the system performance and fairness for five previous schedulers and BLISS with different last level cache sizes (private to each core).

We make two observations. First, the absolute values of weighted speedup increase and maximum slowdown

---

9. We classify an application as having high row-buffer locality if its row-buffer hit rate is greater than 90 percent.

10. Fairness benefits reduce at very high core counts and very low channel counts, since memory bandwidth becomes highly saturated.
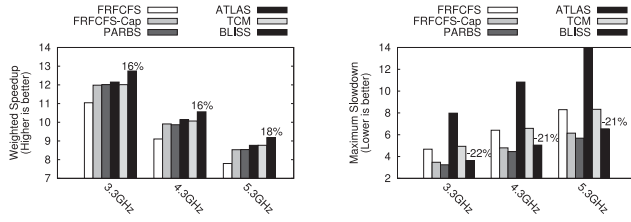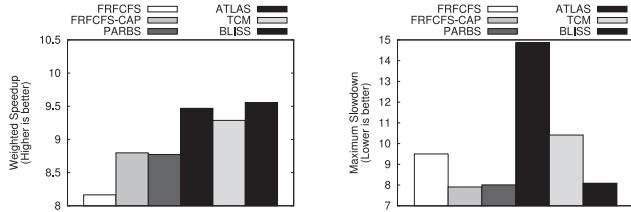
Fig. 21. Sensitivity to core frequency.

Fig. 22. Performance and fairness with a shared cache.

TABLE 3
Perf. Sensitivity to Threshold and Interval

Threshold\ Interval	1,000	10,000	100,000
2	8.76	8.66	7.95
4	8.61	9.18	8.60
8	8.42	9.05	9.24

TABLE 4
Unfairness Sensitivity to Threshold and Interval

Threshold\ Interval	1,000	10,000	100,000
2	6.07	6.24	7.78
4	6.03	6.54	7.01
8	6.02	7.39	7.29

decrease, as the cache size becomes larger for all schedulers, as expected. This is because contention for memory bandwidth reduces with increasing cache capacity, improving performance and fairness. Second, across all the cache capacity points we evaluate, BLISS achieves significant performance and fairness benefits over the best-performing previous schedulers, while approaching close to the fairness of the fairest previous schedulers.

Core frequency. Fig. 21 shows the system performance and fairness for five previous schedulers and BLISS with different CPU core frequencies.

We draw two conclusions. First, as the core frequency increases, the absolute value of weighted speedup decreases. This is because, at lower core frequencies, memory is less of a bottleneck. Hence, applications slow down less due to memory contention from sharing the system with other applications, resulting in higher weighted speedup at lower core frequencies. Second, BLISS achieves significant performance and fairness benefits over the best-performing previous schedulers, while approaching the fairness of the fairest previous schedulers.

Shared caches. Fig. 22 shows system performance and fairness with a 32 MB shared cache (instead of the 512 KB per core private caches used in our other experiments). BLISS achieves 5%/24% better performance/fairness compared to TCM, demonstrating that BLISS is effective in mitigating memory interference in the presence of large shared caches as well.

Prefetching. Prefetching can enable requests to be brought into the last level cache from main memory, potentially mitigating the impact of memory contention. We implemented a stride prefetcher that learns strided access patterns generated from each core and prefetches data from the main memory into the last level cache. We observe that implementing such a prefetcher on top of the baseline FRFCFS policy results in a performance/fairness degradation of 10%/16%. This is because prefetching that is unaware of the current workload access patterns and memory bandwidth utilization in a multicore workload, can cause interference to other applications' requests. However, BLISS still improves performance/fairness by 22%/27% over FRFCFS, on a system with a stride prefetcher. In the future, we propose to study BLISS along with feedback-driven prefetch mechanisms such as feedback directed prefetching [38],

shared-resource-management-aware prefetch policies [39] and coordinated prefetch policies [40].

7.12 Sensitivity to Algorithm Parameters

Tables 3 and 4 show the system performance and fairness respectively of BLISS for different values of the *Blacklisting Threshold* and *Clearing Interval*. Two major conclusions are in order. First, a *Clearing Interval* of 10,000 cycles provides a good balance between performance and fairness. If the blacklist is cleared too frequently (1,000 cycles), interference-causing applications are not deprioritized for long enough, resulting in low system performance. In contrast, if the blacklist is cleared too infrequently, interference-causing applications are deprioritized for too long, resulting in high unfairness. Second, a *Blacklisting Threshold* of 4 provides a good balance between performance and fairness. When *Blacklisting Threshold* is very small, applications are blacklisted as soon as they have very few requests served, resulting in poor interference mitigation since too many applications are blacklisted. On the other hand, when *Blacklisting Threshold* is large, low- and high-memory-intensity applications are not segregated effectively, leading to high unfairness.

7.13 Interleaving and Scheduling Interaction

In this section, we study the impact of the address interleaving policy on the performance and fairness of different schedulers. Our analysis so far has assumed a row-interleaved policy, where data is distributed across channels, banks and rows at the granularity of a row. This policy optimizes for row-buffer locality by mapping a consecutive row of data to the same channel, bank, rank. In this section, we will consider two other interleaving policies, cache block interleaving and sub-row interleaving.

Interaction with cache block interleaving. In a cache-block-interleaved system, data is striped across channels, banks and ranks at the granularity of a cache block. Such a policy optimizes for bank level parallelism, by distributing data at a small (cache block) granularity across channels, banks and ranks.

Fig. 23 shows the system performance and fairness of FRFCFS with row interleaving (FRFCFS-Row), as a comparison point, five previous schedulers and BLISS with cache block interleaving. We draw three observations. First, system performance and fairness of the baseline FRFCFS scheduler improve significantly with cache block interleaving,
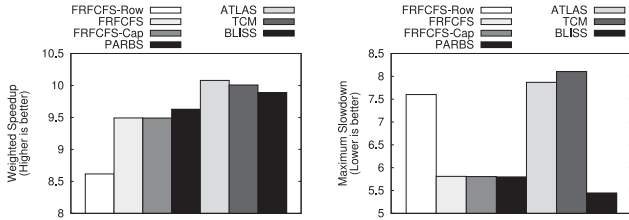
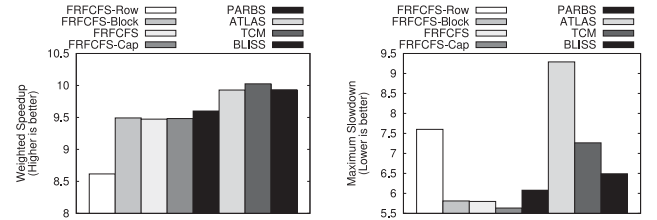Fig. 23. Scheduling and cache block interleaving.

Fig. 24. Scheduling and sub-row interleaving.

compared to performance/fairness with row interleaving. This is because cache block interleaving enables more requests to be served in parallel at the different channels and banks, by distributing data across channels and banks at the small granularity of a cache block. Hence, most applications, and particularly, applications that do not have very high row-buffer locality benefit from cache block interleaving.

Second, as expected, application-aware schedulers such as ATLAS and TCM achieve the best performance among previous schedulers, by means of prioritizing requests of applications with low memory intensities. However, PARBS and FRFCFS-Cap do not improve fairness over the baseline, in contrast to our results with row interleaving. This is because cache block interleaving already attempts to provide fairness by increasing the parallelism in the system and enabling more requests from across different applications to be served in parallel, thereby reducing unfair applications slowdowns. More specifically, requests that would be row-buffer hits to the same bank, with row interleaving, are now distributed across multiple channels and banks, with cache block interleaving. Hence, applications' propensity to cause interference reduces, providing lower scope for request capping based schedulers such as FRFCFS-Cap and PARBS to mitigate interference. Third, BLISS achieves within 1.3 percent of the performance of the best performing previous scheduler (ATLAS), while achieving 6.2 percent better fairness than the fairest previous scheduler (PARBS). BLISS effectively mitigates interference by regulating the number of consecutive requests served from high-memory-intensity applications that generate a large number of requests, thereby achieving high performance and fairness.

Interaction with sub-row interleaving. While memory scheduling has been a prevalent approach to mitigate memory interference, previous work has also proposed other solutions, as we describe in Section 8. One such previous work by Kaseridis et al. [10] proposes *minimalist open page*, an interleaving policy that distributes data across channels, ranks and banks at the granularity of a sub-row (partial row), rather than an entire row, exploiting both row-buffer locality and bank-level parallelism. We examine BLISS' interaction with such a sub-row interleaving policy.

Fig. 24 shows the system performance and fairness of FRFCFS with row interleaving (FRFCFS-Row), FRFCFS with cache block interleaving (FRFCFS-Block) and five previously proposed schedulers and BLISS, with sub-row interleaving (when data is striped across channels, ranks and banks at the granularity of four cache blocks). Three observations are in order. First, sub-row interleaving provides significant benefits over row interleaving, as can be observed for FRFCFS (and other scheduling policies by comparing with Fig. 4). This is because sub-row interleaving enables applications to

exploit both row-buffer locality and bank-level parallelism, unlike row interleaving that is mainly focused on exploiting row-buffer locality. Second, sub-row interleaving achieves similar performance and fairness as cache block interleaving. We observe that this is because cache block interleaving enables applications to exploit parallelism effectively, which makes up for the lost row-buffer locality from distributing data at the granularity of a cache block across all channels and banks. Third, BLISS achieves close to the performance (within 1.5 percent) of the best performing previous scheduler (TCM), while reducing unfairness significantly and approaching the fairness of the fairest previous schedulers. One thing to note is that BLISS has higher unfairness than FRFCFS, when a sub-row-interleaved policy is employed. This is because the capping decisions from sub-row interleaving and BLISS could collectively restrict high-row-buffer locality applications to a large degree, thereby slowing them down and causing higher unfairness. Co-design of the scheduling and interleaving policies to achieve different goals such as performance/fairness is an important area of future research. We conclude that a BLISS-like scheduler, with its high performance and low complexity is a significantly better alternative to schedulers such as ATLAS/TCM in the pursuit of such scheduling-interleaving policy co-design.

8 RELATED WORK

To our knowledge, BLISS is the first memory scheduler design that attempts to optimize, at the same time, for high performance, fairness and low complexity, which are three competing yet important goals. The closest previous works to BLISS are other memory schedulers. We have already compared BLISS both qualitatively and quantitatively to previously proposed memory schedulers, FRFCFS [17], [23], PARBS [4], [5], ATLAS [6], TCM [7] and criticality-aware memory scheduling [8], which have been designed to mitigate interference in a multicore system. Other previous schedulers [1], [2], [3] have been proposed earlier that PARBS, ATLAS and TCM have been shown to outperform [4], [6], [7].

Previous works also proposed models, MISE [12], ASM [13], to estimate the impact of inter-application interference in terms of slowdowns in order to enable a substrate for providing QoS. BLISS, on the other hand, focuses on simplicity, fairness and performance. Hence, these two approaches target different problems. However, principles from both of these approaches can be combined as part of future work.

Parallel Application Memory Scheduling (PAMS) [41] tackles the problem of mitigating interference between different threads of a multithreaded application. Principles from BLISS can be employed in this context to identify and deprioritize interference-causing threads, thereby mitigating

interference experienced by vulnerable threads/applications. For instance, information on how much progress different threads are making towards a synchronization point can be combined along with vulnerability to interference to prioritize among threads. The idea of grouping can be extended to first classify threads into critical versus non-critical depending on how much progress they have made towards a synchronization point. Within each of these categories, threads can then be classified as vulnerable versus interference-causing depending on the number of consecutive requests served from the different threads. In the case of homogeneous threads, classifying threads as vulnerable versus interference-causing based on monitoring the number of consecutive requests from a thread can effectively serve as a shuffling mechanism, preventing the same thread's requests from being served and hogging memory bandwidth.

Staged Memory Scheduling (SMS) [42] attempts to mitigate interference between the CPU and GPU in CPU-GPU systems. The goals of SMS, designed for CPU-GPU systems, are very different from those of BLISS. SMS seeks to achieve high GPU performance, while still achieving reasonable CPU performance. Furthermore, the SMS work also focuses on providing a knob to trade off CPU and GPU performance. The goals of BLISS, on the other hand, are to improve system performance and fairness in a CPU-only multicore system. Hence, we do not quantitatively compare to SMS. In fact, quantitative comparisons in the SMS paper (Section 6.6, Fig. 12) show that SMS does not perform as well as ATLAS (and in some cases, TCM), in the context of CPU-only multicore systems, as it is optimized and built for different goals. DASH [43], a recently proposed memory scheduler, tackles the problem of memory scheduling in the context of systems with CPUs, GPUs and hardware accelerators. Similarly to SMS, the goals and principles of DASH are different from those of BLISS. Principles from BLISS and SMS/DASH can potentially be combined to achieve better interference mitigation in heterogeneous systems with CPUs, GPUs and hardware accelerators.

FIRM [44] proposes request scheduling mechanisms to tackle the problem of heavy write traffic in persistent memory systems. BLISS can be combined with FIRM's write handling mechanisms to achieve better fairness in persistent memory systems. Complexity effective memory access scheduling [45] attempts to achieve the performance of FRFCFS using a First Come First Served scheduler in GPU systems, by preventing row-buffer locality from being destroyed when data is transmitted over the on-chip network. Their proposal is complementary to ours. BLISS could be combined with such a scheduler design to prevent threads from hogging the row-buffer and banks.

While memory scheduling is a major solution direction towards mitigating interference, previous works have also explored other approaches such as address interleaving [10], memory bank/channel partitioning [9], [46], [47], [48], source throttling [11], [49], [50], [51], [52], [53], [54], [55] and thread scheduling [15], [16], [56], [57] to mitigate interference.

Sub-row interleaving. Kaseridis et al. [10] propose minimalist open page, a data mapping policy that interleaves data at the granularity of a sub-row across channels and banks such that applications with high row-buffer locality are prevented from hogging the row buffer, while still preserving some amount of row-buffer-locality. We study the interactions of BLISS with minimalist open page in Section 7.13 showing BLISS' benefits on a sub-row interleaved memory system.

Memory channel/bank partitioning. Previous works [9], [46], [47], [48] propose techniques to mitigate inter-application interference by partitioning channels/banks among applications such that the data of interfering applications are mapped to different channels/banks. Our approach is complementary to these schemes and can be used in conjunction with them to achieve more effective interference mitigation.

Source throttling. Source throttling techniques (e.g., [11], [49], [50], [51], [53], [54], [55]) propose to throttle the memory request injection rates of interference-causing applications at the processor core itself rather than regulating an application's access behavior at the memory, unlike memory scheduling, partitioning or interleaving. BLISS is complementary to source throttling and can be combined with it to achieve better interference mitigation.

OS thread scheduling. Previous works (e.g., [15], [16], [57]) propose to mitigate shared resource contention by co-scheduling threads that interact well and interfere less at the shared resources. Such a solution relies on the presence of enough threads with such symbiotic properties, whereas our proposal can mitigate memory interference even if interfering threads are co-scheduled. Furthermore, such thread scheduling policies and BLISS can be combined in a synergistic manner to further improve system performance and fairness. Other techniques to map applications to cores to mitigate memory interference, such as [56], can be combined with BLISS.

9 CONCLUSION

We introduce the Blacklisting memory scheduler, a new and simple approach to memory scheduling in systems with multiple threads. We observe that the per-application ranking mechanisms employed by previously proposed application-aware memory schedulers incur high hardware cost, cause high unfairness, and lead to high scheduling latency to the point that the scheduler cannot meet the fast command scheduling requirements of state-of-the-art DDR protocols. BLISS overcomes these problems based on the key observation that it is sufficient to group applications into only two groups, rather than employing a total rank order among all applications. Our evaluations across a variety of workloads and systems demonstrate that BLISS has better system performance and fairness than previously proposed ranking-based schedulers, while incurring significantly lower hardware cost and latency in making scheduling decisions. We conclude that BLISS, with its low complexity, high system performance and high fairness, can be an efficient and effective memory scheduling substrate for current and future multicore and multithreaded systems. We released the source code of BLISS at [28] to aid future research building upon this work.

ACKNOWLEDGMENTS

REFERENCES

[1] K. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2006, pp. 208–222.

[2] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *Proc. 16th USENIX Security Symp. USENIX Security Symp.*, 2007, pp. 18:1–18:18

[3] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2007, pp. 146–160.

[4] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *Proc. 35th Annu. Int. Symp. Comput. Archit.*, 2008, pp. 63–74.

[5] T. Moscibroda and O. Mutlu, "Distributed order scheduling and its application to multi-core DRAM controllers," in *Proc. 27th ACM Symp. Principles Distrib. Comput.*, 2008, pp. 365–374.

[6] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *Proc. IEEE 16th Int. Symp. High Perform. Comput. Archit.*, 2010, pp. 1–12.

[7] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: exploiting differences in memory access behavior," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2010, pp. 65–76.

[8] S. Ghose, H. Lee, and J. F. Martínez, "Improving memory scheduling via processor-side load criticality information," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 84–95.

[9] S. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via Application-aware memory channel partitioning," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2011, pp. 374–385.

[10] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2011, pp. 24–35.

[11] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems," in *Proc. 15th Ed. ASPLOS Archit. Support Program. Lang. Oper. Syst.*, 2010, pp. 335–346.

[12] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit.*, 2013, pp. 639–650.

[13] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *Proc. 48th Int. Symp. Microarchit.*, 2015, pp. 62–75.

[14] H. Kim, D. D. Niz, B. Andersson, M. Klien, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in COTS-based multi-core systems," in *Proc. Real-Time Embedded Technol. Appl. Symp.*, 2014, pp. 145–154.

[15] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Proc. 38th Annu. Int. Symp. Comput. Archit.*, 2011, pp. 283–294.

[16] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proc. 15th Ed. ASPLOS Archit. Support Program. Lang. Oper. Syst.*, 2010, pp. 129–142.

[17] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proc. 27th Annu. Int. Symp. Comput. Archit.*, 2000, pp. 128–138.

[18] *JEDEC,* Standard No. 79-3. DDR3 SDRAM STANDARD, 2010.

[19] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarray-level parallelism (SALP) in DRAM," in *Proc. 39th Annu. Int. Symp. Comput. Archit.*, 2012, pp. 368–379.

[20] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-latency DRAM: A low latency and low cost DRAM architecture," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit.*, 2013, pp. 615–626.

[21] D. Lee, K. Yoongu, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, "Adaptive-latency DRAM: Optimizing DRAM timing for the common-case," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 489–501.

[22] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. Gibbons, M. Kozuch, and T. Mowry, "RowClone: Fast and efficient In-DRAM copy and initialization of bulk data," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2013, pp. 185–197.

[23] W. Zuravleff and T. Robinson, "Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order," Patent 5 630 096, 1997.

[24] SPEC. (2006). SPEC CPU2006 [Online]. Available: http://www.spec.org/spec2006

[25] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," *IEEE Comput. Archit. Lett.*, vol. PP, no. 99, pp. 1–1, 2015.

[26] CMU SAFARI Research Group. (2015). Ramulator [Online]. Available: https://github.com/CMU-SAFARI/ramulator

[27] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2005, pp. 190–200.

[28] CMU SAFARI Research Group. (2015). MemSchedSim [Online]. Available: https://github.com/CMU-SAFARI/MemSchedSim

[29] Micron, "2Gb: x4, x8, x16, DDR3 SDRAM," 2012.

[30] NASA advanced supercomputing division, *NAS Parallel Benchmark Suite* [Online]. Available: https://www.nas.nasa.gov/publications/npb.html.

[31] R. Das, O. Mutlu, T. Moscibroda, and C. Das, "Application-aware prioritization mechanisms for on-chip networks," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 280–291.

[32] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, May/Jun. 2008.

[33] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proc. 9th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2000, pp. 234–244.

[34] H. Vandierendonck and A. Seznec, "Fairness metrics for multi-threaded processors," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 4–7, Jan. 2011.

[35] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in SMT processors," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2001, pp. 164–171.

[36] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *Proc. 21st Int. Conf. Parallel Archit. Compilation Techn.*, 2012, pp. 355–366.

[37] *JEDEC,* Standard No. 79-4. DDR4 SDRAM STANDARD, 2012.

[38] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proc. IEEE 13th Int. Symp. High Perform. Comput. Archit.*, 2007, pp. 63–74.

[39] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware shared resource management for multi-core systems," in *Proc. 38th Annu. Int. Symp. Comput. Archit.*, 2011, pp. 141–152.

[40] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 316–326.

[41] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel application memory scheduling," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2011, pp. 362–373.

[42] R. Ausavarungnirun, K. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems," in *Proc. 39th Annu. Int. Symp. Comput. Archit.*, 2012, pp. 416–427.

[43] H. Usui, L. Subramanian, K. Chang, and O. Mutlu, "DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators," *ACM Trans. Archit. Code. Optim.*, vol. 12, no. 3, pp. 65:1–65:28, Jan. 2016.

[44] J. Zhao, O. Mutlu, and Y. Xie, "FIRM: Fair and high-performance memory control for persistent memory systems," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2014, pp. 153–165.

[45] G. Yuan, A. Bakhoda, and T. M. Aamodt, "Complexity effective memory access scheduling for many-core accelerator architectures," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 34–44.

[46] M. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, "Balancing DRAM locality and parallelism in shared memory CMP systems," in *Proc. IEEE 18th Int. Symp. High Perform. Comput. Archit.*, 2012, pp. 1–12.

[47] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *Proc. 21st Int. Conf. Parallel Archit. Compilation Techn.*, 2012, pp. 367–376.

[48] M. Xie, D. Tong, K. Huang, and X. Cheng, "Improving system throughput and fairness simultaneously in shared memory CMP systems via dynamic bank partitioning," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 344–355.

[49] M. Thottethodi, A. Lebeck, and S. Mukherjee, "Self-tuned congestion control for multiprocessor networks," in *Proc. 7th Int. Symp. High-Perform. Comput. Archit.*, 2001, p. 107.

[50] E. Baydal, P. Lopez, and J. Duato, "A family of mechanisms for congestion control in wormhole networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 9, pp. 772–784, Sep. 2005.

[51] K. Chang, R. Ausavarungnirun, C. Fallin, and O. Mutlu, "HAT: Heterogeneous adaptive throttling for on-chip networks," in *Proc. IEEE Int. Symp. Comput. Archit. High Perform. Comput.*, 2012, pp. 9–18.

[52] R. Ausavarungnirun, C. Fallin, X. Yu, K. Chang, G. Nazario, R. Das, G. Loh, and O. Mutlu, "Design and evaluation of hierarchical rings with deflection routing," in *Proc. IEEE 26th Int. Symp. Comput. Archit. High Perform. Comput.*, 2014, pp. 230–237.

[53] G. Nychis, C. Fallin, T. Moscibroda, and O. Mutlu, "On-chip networks from a networking perspective: Congestion and scalability in many-core interconnects," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit. Protocols Comput. Commun.*, 2012, pp. 407–418.

[54] G. Nychis, C. Fallin, T. Moscibroda, and O. Mutlu, "Next generation on-chip networks: What kind of congestion control do we need?" in *Proc. 9th ACM SIGCOMM Workshop Hot Topics Netw.*, 2010, pp. 12:1–12:6.

[55] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing GPU concurrency in heterogeneous architectures," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2014, pp. 114–126.

[56] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi, "Application-to-core mapping policies to reduce memory system interference in multi-core systems," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit.*, 2013, pp. 107–118.

[57] H. Wang, C. Isci, L. Subramanian, J. Choi, D. Qian, and O. Mutlu, "A-DRM: Architecture-aware distributed resource management of virtualized clusters," in *Proc. 11th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2015, pp. 93–106.

[58] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The blacklisting memory scheduler: Achieving high performance and fairness at low cost," in *Proc. 32nd IEEE Int. Conf. Comput. Des.*, 2014, pp. 8–15.

Lavanya Subramanian received the master's and PhD degrees in electrical and computer engineering from Carnegie Mellon University. She is a research scientist at Intel Labs. She is interested in the general area of computer architecture, with specific focus on providing predictable performance and Quality of Service in the presence of shared resource interference.

Donghyuk Lee is currently working toward the PhD degree at Carnegie Mellon University. He is interested in high-performance DRAM-based memory systems, especially from the perspective of achieving low latency and high bandwidth. He was a DRAM circuit designer at Samsung before he started his PhD.

Vivek Seshadri received the bachelor of technology in computer science from the Indian Institute of Technology, Madras in 2009. He is currently working toward the PhD degree at Carnegie Mellon University. He is advised by Prof. Todd C. Mowry and Prof. Onur Mutlu. His research interests include DRAM microarchitecture, cache management, and computer architecture/systems in general.

Harsha Rastogi received the bachelors in electrical and electronics engineering from BITS, Pilani, India. She is currently working toward the masters degree in the ECE Department of Carnegie Mellon University. Her interests lie in computer architecture, storage and distributed systems. Before she started at Carnegie Mellon, she was a design engineer at Broadcom.

Onur Mutlu received the BS degrees in computer engineering and psychology from the University of Michigan, Ann Arbor, and the MS and PhD degrees from the University of Texas at-Austin. He is the Strecker early career professor at Carnegie Mellon University. His research interests are in computer architecture and systems, especially in the interactions between languages, operating systems, compilers, and microarchitecture, with a current major focus on memory systems. He previously worked at Microsoft Research from 2006 to 2009, Intel Corporation, and Advanced Micro Devices. He received the IEEE Computer Society Young Computer Architect Award, Intel Early Career Faculty Award, faculty partnership awards from various companies, and a number of best paper and top pick paper recognitions. For more information, please see his webpage at http://www.ece.cmu.edu/~omutlu.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.