The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost

Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, Onur Mutlu
Carnegie Mellon University
{lsubrama,donghyu1,visesh,harshar,onur}@cmu.edu

***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, ranking applications individually with a total order based on memory access characteristics leads to high hardware cost and complexity. Second, ranking can unfairly slow down applications that are at the bottom of the ranking stack. 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 cause interference and another containing applications vulnerable to 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 – an application that has a large number of consecutive requests served is dynamically classified as interference-causing.

We evaluate BLISS across a wide variety of workloads and system configurations and compare its performance and complexity with five state-of-the-art memory schedulers. Our evaluations show that BLISS achieves 5% better system performance and 25% 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%, respectively).

I. 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 [24, 26, 27, 28] 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 [24].

Prior works proposed different solution approaches to mitigate inter-application interference, with the goal of improving system performance and fairness [7, 10, 14, 15, 16, 25, 26, 27, 33, 36, 37]. A prevalent solution direction is application-aware memory request scheduling [15, 16, 26, 27, 33]. The basic idea behind 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 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 VII-A, TCM [16], a state-of-the-art application-aware scheduler is 8x slower and 1.8x larger than a commonly-employed application-unaware scheduler, FRFCFS [30]. Second, 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* 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 (as done in prior works), 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. Although one prior work, TCM [16], proposed to group applications based on memory intensity, TCM still ranks applications individually within each group and enforces the total rank order during scheduling. Our approach overcomes the two major problems with such ranking-based schedulers. First, separating applications into only two groups, as opposed to employing ranking based on a total order of applications, significantly reduces hardware complexity. 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.

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. Our approach to classifying applications is simpler to implement than prior approaches (e.g., [15, 16, 27]) that use more complicated metrics such as memory intensity, row-buffer locality, bank-level paral-

lelism or long-term memory service as proxies for vulnerability to interference.

Mechanism Overview. Based on these two observations, our mechanism, the Blacklisting Memory Scheduler (BLISS), 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.
- We propose the Blacklisting memory scheduler (BLISS), which achieves high system performance and fairness while incurring low hardware cost and complexity. The key idea is to separate applications into only two groups, *vulnerable-to-interference* and *interference-causing*, and deprioritize the latter during scheduling, rather than ranking individual applications with a total order based on their access characteristics.
- We provide a comprehensive complexity analysis of five previously proposed memory schedulers, comparing their critical path latency and area via RTL implementations. 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 [16].
- We evaluate BLISS against five previously-proposed memory schedulers in terms of system performance and fairness across a wide range of workloads. Our results show that BLISS achieves 5% better system performance and 25% better fairness than the best previous scheduler, TCM [16].

II. 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.

A. 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 an 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 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 [11]. For more detail, please see [17, 18].

B. Memory Scheduling

State-of-the-art memory controllers employ a memory scheduling policy called First Ready First Come First Served (FRFCFS) [30, 38] 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) [24, 26]. Previous work [15, 16, 26, 27] 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 VII.

Mutlu and Moscibroda propose PARBS [27], an application-aware memory scheduler that batches the oldest requests from applications and prioritizes the batched requests in order to prevent starvation. 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.

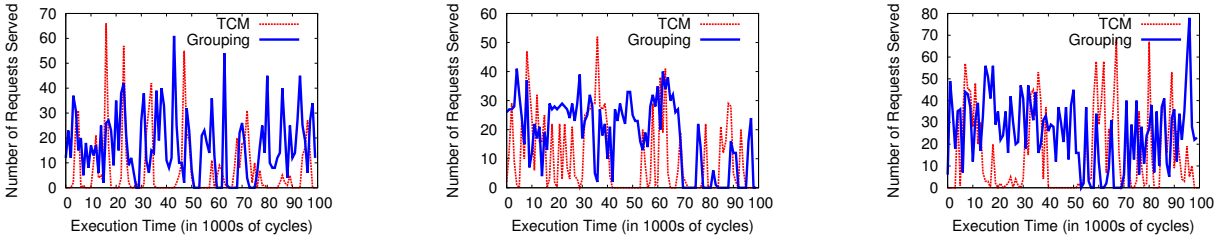
Kim et al. in [15] observe that applications that receive low memory service tend to experience interference from applications that receive high memory service. Based on this observation, they propose ATLAS, an application-aware memory scheduling policy that ranks individual applications based on the amount of long-term memory service each receives and prioritizes applications that receive low memory service.

Thread cluster memory scheduling (TCM) [16] ranks individual applications by memory intensity such that low-memory-intensity applications are prioritized over high-memory-intensity applications. Kim et al. [16] 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. In order to mitigate 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. [10] 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.

C. 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 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 [15, 16, 27]. Scheduling requests based on a total rank order incurs high hardware complexity, as we demonstrate in Section VII-A, slowing down the memory scheduler significantly (by 8x for TCM compared to FRFCFS), while also increasing its area (by 1.8x). 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, 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



(a) astar (b) hmmer (c) lbm

Fig. 1: Request service distribution over time with TCM and Grouping schedulers

is at the bottom of the ranking stack, as we show in Section III.

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* (*BLISS*) based on two new observations described in the next section.

III. Key Observations

Several previous 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 II-C. We seek to 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, is sufficient to mitigate inter-application interference.*

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 [16], which clusters applications into two groups and employs a total rank order within each group, with a simple scheduling mechanism (*Grouping*) that simply groups applications into two, 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. Figure 1 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 [2], using these two schedulers.[1] These three applications are run along with other applications in a simulated 24-core 4-channel system.[2]

Figure 1 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 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 Figure 1 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 ranking.

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 Figure 1. However, the request service behavior of *Grouping* is less spiky than with 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 ranking 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 Figure 2 (across 80 24-core workloads on a simulated 4-channel system).

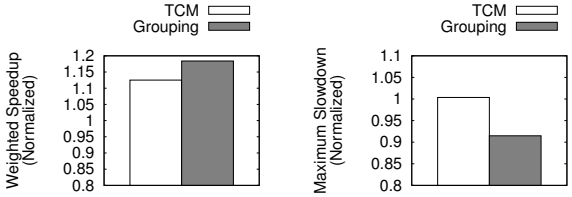


Fig. 2: Performance and fairness of Grouping

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 Figure 2), 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.

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.*

Previous work attempted to perform grouping, along with ranking, to mitigate interference. Specifically, TCM [16] 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

[1] All these three applications are in the high-memory-intensity group.

[2] See Section VI for our methodology.

remaining applications into 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 Figure 2), it i) *can still cause unfair slowdowns due to classifying applications into groups at coarse time granularities, which is especially dangerous if application behavior changes during an interval*, and ii) *incurs additional hardware overhead and scheduling latency to compute and rank by long-term memory intensity and total memory bandwidth usage*.

We propose to perform grouping using a significantly simpler 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 1000s 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 VII-B).

IV. Mechanism

In this section, we present the details of our Blacklisting memory scheduler (BLISS) that employs a simple grouping scheme motivated by our key observations from Section III. The basic idea behind BLISS is to observe the number of consecutive requests served from an application over a short time interval and blacklist applications that have a relatively large number of consecutive requests served. The blacklisted (interference-causing) and non-blacklisted (vulnerable-to-interference) applications are thus separated into two different groups. The memory scheduler then prioritizes the non-blacklisted group over the blacklisted group. The two main components of BLISS are i) the blacklisting mechanism and ii) the memory scheduling mechanism that schedules requests based on the blacklisting mechanism. We describe each in turn.

A. 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*)[3], 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.

[3]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.

If the *#Requests Served* exceeds a *Blacklisting Threshold* (set to 4 in 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* (set to 10000 cycles in our evaluations).

B. 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 and then older requests, for forward progress.

V. 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.

A. 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)

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

B. 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 quantitative evaluation of the hardware area cost and latency of implementing BLISS and previously proposed memory schedulers, in Section VII-A.

VI. Methodology

A. System Configuration

We model the DRAM memory system using a cycle-level in-house DDR3-SDRAM simulator. The simulator was validated against Micron's behavioral Verilog model [22] and DRAMSim2 [31]. This DDR3 simulator is integrated with a cycle-level in-house simulator that models out-of-order execution cores, driven by a Pin [20] tool at the frontend, Each core has a private cache of 512 KB size. 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 interference on application performance. We also present results with shared caches in Section VII-G. Table I provides more details on our simulated system. We perform most of our studies on a system with 24 cores and 4 channels. We provide a sensitivity analysis for a wide range of core and channel counts, in Section VII-E. Each channel has one rank and each rank has eight banks. We stripe data across channels and banks at the granularity of a row.

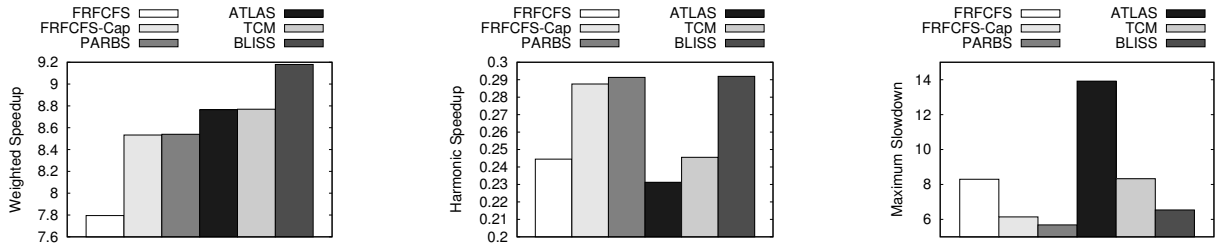


Fig. 3: System performance and fairness of BLISS compared to previous schedulers

Processor	16-64 cores, 5.3GHz, 3-wide issue, 8 MSHRs, 128-entry instruction window
Last-level cache	64B cache-line, 16-way associative, 512KB private cache-slice per core
Memory controller	128-entry read/write request queue per controller
Memory	Timing: DDR3-1066 (8-8-8) [23] Organization: 1-8 channels, 1 rank-per-channel, 8 banks-per-rank, 8 KB row-buffer

TABLE I: Configuration of the simulated system

B. Workloads

We perform our main studies using 24-core multiprogrammed workloads made of applications from the SPEC CPU2006 suite [2], TPC-C, Matlab and the NAS parallel benchmark suite [1].[4] We construct 80 workloads with a range of memory intensities using random combinations of benchmarks. We simulate each workload for 100 million representative cycles.

C. Metrics

We quantitatively compare BLISS with previous memory schedulers in terms of system performance, fairness and complexity. We use the weighted speedup [6, 9, 32] metric to measure system performance. We use the maximum slowdown metric [6, 15, 16, 35] to measure unfairness. We report the harmonic speedup metric [21] as another measure of system performance. The harmonic speedup metric also serves as a measure of balance between system performance and fairness [21]. We report area in $micrometer^2$ (um^2) and scheduler critical path latency in nanoseconds (ns) as measures of complexity.

D. 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.

E. Mechanism Parameters

For BLISS, we use a value of four for *Blacklisting Threshold*, and a value of 10000 cycles for *Clearing Interval*. These values provide a good balance between performance and fairness, as we observe from our sensitivity studies in Section VII-F. 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 1000 cycles for TCM.

VII. Evaluation

We compare BLISS with five previously proposed memory schedulers, FRFCFS, FRFCFS with a cap (FRFCFS-Cap), 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 [26]. Figure 3 shows the average system performance (weighted speedup and harmonic speedup) and unfairness (maximum slowdown) across all our workloads. Figure 4 shows a Pareto plot of weighted speedup and maximum slowdown. We make three major observations. First, BLISS achieves 5% better weighted speedup, 25% better maximum slowdown and 19% better harmonic speedup than TCM, the best performing previous scheduler (in terms of weighted speedup), while reducing the critical path and area by 79% and 43% respectively (as we will show in Section VII-A). Therefore, we conclude that BLISS achieves both high system performance and fairness, at low hardware cost and complexity.

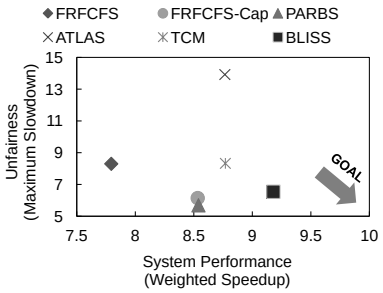


Fig. 4: Pareto plot of system performance and fairness

Second, BLISS significantly outperforms all these five previous schedulers in terms of system performance, however, it has 10% 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 request reordering have lower performance. As a result, PARBS has 8% lower weighted speedup than BLISS. Furthermore, PARBS has a 6.5x longer critical path and ~2x greater area than BLISS, as we will show in Section VII-A. 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% higher unfairness than FRFCFS-Cap, but 8% higher performance than FRFCFS-Cap. This is because FRFCFS-Cap breaks long row hit chains that could potentially delay other applications' requests when a naive FRFCFS scheduler is employed. Hence, FRFCFS-Cap only restricts the length of 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, thereby achieving lower unfairness than BLISS. However, restricting *only* the on-going streak rather than blacklisting an interfering application causes higher interference to other applications, degrading system performance compared to BLISS. Furthermore,

[4] Each benchmark is single threaded.

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 system performance (weighted speedup) than FRFCFS-Cap, while slightly trading off fairness.

A. Hardware Complexity

Figures 5 and 6 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 11x/5.3x/8.1x and 2.4x/1.7x/1.8x respectively, compared to FRFCFS and FRFCFS-Cap, whereas BLISS increases latency and area by only 1.7x and 3.2% over FRFCFS/FRFCFS-Cap.[5] Second, PARBS, ATLAS and TCM cannot meet the stringent worst-case timing requirements posed by the DDR3 and DDR4 standards [11, 12]. 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 the highest released frequency for DDR4, 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 technologies. Techniques like pipelining could potentially be employed to reduce the critical path latency. 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.

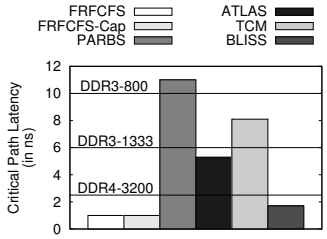


Fig. 5: Critical path: BLISS vs. previous schedulers

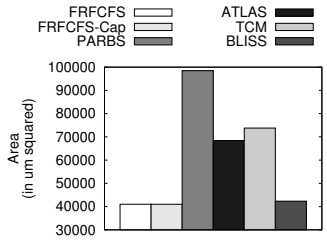


Fig. 6: Area: BLISS vs. previous schedulers

B. Comparison with TCM's Clustering Mechanism

Figure 7 shows the system performance and fairness of BLISS, TCM and TCM's clustering mechanism (TCM-Cluster). TCM-Cluster 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 III). 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 on the same channel).

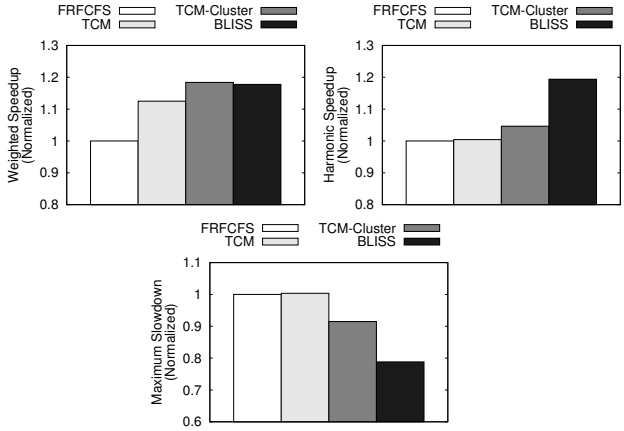


Fig. 7: Comparison with TCM's clustering mechanism

C. Comparison with Criticality-Aware Scheduling

We compare the system performance and fairness of BLISS with those of criticality-aware memory schedulers [10]. 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. [10] evaluate prioritizing load requests based on both maximum and total stall times caused by loads instructions in the past. Figure 8 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 [10]. 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 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.

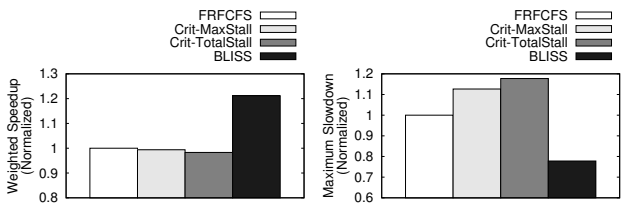


Fig. 8: Comparison with criticality-aware scheduling

[5]The area numbers are for the lowest value of critical path latency that the scheduler is able to meet.

D. Effect of Workload Memory Intensity

Figure 9 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.[6] 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 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 all intensity categories.

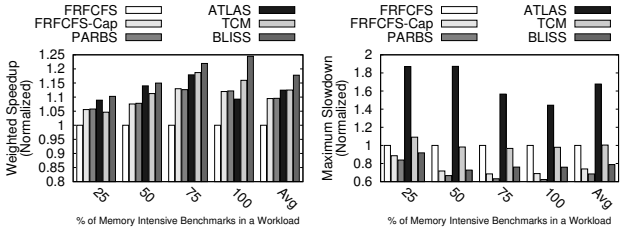


Fig. 9: Sensitivity to workload memory intensity

E. Sensitivity to System Parameters

Figures 10 and 11 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. We draw two major conclusions. First, BLISS achieves higher system performance and lower unfairness than all the other scheduling policies (except PARBS, 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.

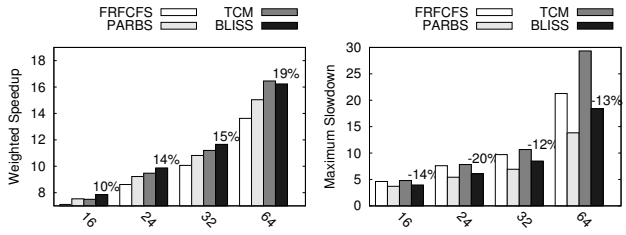


Fig. 10: Sensitivity to number of cores

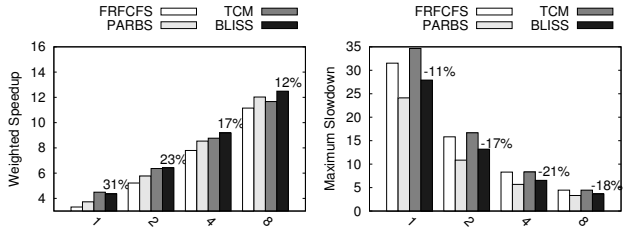


Fig. 11: Sensitivity to number of channels

Second, BLISS' system performance benefit (as indicated by the percentages on top of bars, over FRFCFS) 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.[7]

F. Sensitivity to Algorithm Parameters

Tables II and III show the system performance and fairness of BLISS for different values of the *Blacklisting Threshold* and *Clearing Interval* respectively, across 40 workloads. Three major conclusions are in order. First, a *Clearing Interval* of 10000 cycles provides a good balance between performance and fairness. If the blacklist is cleared too frequently (1000 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 as many applications are quickly blacklisted. On the other hand, when *Blacklisting Threshold* is large, low and high memory-intensity applications are not segregated effectively.

Threshold \ Interval	1000	10000	100000
2	8.76	8.66	7.95
4	8.61	9.18	8.60
8	8.42	9.05	9.24

TABLE II: Perf. sensitivity to threshold and interval

Threshold \ Interval	1000	10000	100000
2	6.07	6.24	7.78
4	6.03	6.54	7.01
8	6.02	7.39	7.29

TABLE III: Unfairness sensitivity to threshold and interval

G. Shared Caches

Table IV 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 shared caches as well.

Metric	FRFCFS	TCM	BLISS
Weighted Speedup (Normalized)	1	1.13	1.18
Maximum Slowdown (Normalized)	1	0.99	0.75

TABLE IV: Performance and fairness with a shared cache

VIII. Related Work

Memory Scheduling: The closest previous works to BLISS are other memory scheduling techniques. We have already compared BLISS both qualitatively and quantitatively to previously proposed memory schedulers, FRFCFS [30, 38], PARBS [27], ATLAS [15], TCM [16] and criticality-aware memory scheduling [10], which have been designed to mitigate interference in a multicore system. Parallel Application Memory Scheduling (PAMS) [8] tackles the problem of mitigating interference between different threads of a multithreaded application, while Staged Memory Scheduling (SMS) [3] attempts to mitigate interference between the CPU and GPU in CPU-GPU systems. Principles from BLISS can be employed in both of these contexts to identify and deprioritize interference-causing threads, thereby mitigating interference experienced by vulnerable threads/applications.

[6] We classify applications with MPKI less than 5 as low-memory-intensity and the rest as high-memory-intensity.

[7] Fairness benefits reduce at very high core counts and very low channel counts, since memory bandwidth becomes highly saturated.

While memory scheduling is a major solution direction towards mitigating interference, previous works have also explored other approaches such as interleaving [14], memory bank/channel partitioning [13, 19, 25], source throttling [4, 7, 29] and thread scheduling [34, 37] to mitigate interference.

Subrow Interleaving: Kaseridis et al. [14] 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. Memory scheduling and interleaving can be employed in a complementary manner, as illustrated in [14]. Our evaluations also show a 7% performance improvement when BLISS is implemented on top of minimalist open page.

Memory Channel/Bank Partitioning: Previous works [13, 19, 25] 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., [4, 7, 29]) 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 [34, 37] 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 [5], can be combined with BLISS.

IX. Conclusion

We introduce the Blacklisting memory scheduler (BLISS), a new and simple approach to memory scheduling in multicore systems. 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 different 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 systems.

Acknowledgments

We thank the reviewers for their valuable suggestions. We thank Brian Prasky and Viji Srinivasan from IBM for their feedback and helpful comments at various stages of the project. We acknowledge the generous support from our industrial partners: IBM, Intel, Microsoft, Qualcomm, Samsung, VMware. This research was partially funded by NSF grants (CAREER Award CCF 0953246, CCF 1212962, and CNS 1065112), Intel Science and Technology Center for Cloud Computing, and the Semiconductor Research Corporation. Lavanya Subramanian is partially supported by a John and Claire Bertucci fellowship.

References

[1] *NAS Parallel Benchmark Suite*, http://www.nas.nasa.gov/publications/npb.html.

[2] *SPEC CPU2006*, http://www.spec.org/spec2006.

[3] R. Ausavarungnirun *et al.*, "Staged Memory Scheduling: Achieving high performance and scalability in heterogeneous systems," in *ISCA*, 2012.

[4] K. Chang, R. Ausavarungnirun, C. Fallin, and O. Mutlu, "HAT: Heterogeneous adaptive throttling for on-chip networks," in *SBAC-PAD*, 2012.

[5] R. Das *et al.*, "Application-to-core mapping policies to reduce memory system interference in multi-core systems," in *HPCA*, 2014.

[6] R. Das, O. Mutlu, T. Moscibroda, and C. Das, "Application-aware prioritization mechanisms for on-chip networks," in *MICRO*, 2009.

[7] 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 *ASPLOS*, 2010.

[8] E. Ebrahimi *et al.*, "Parallel application memory scheduling," in *MICRO*, 2011.

[9] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, no. 3, 2008.

[10] S. Ghose, H. Lee, and J. F. Martínez, "Improving memory scheduling via processor-side load criticality information," in *ISCA*, 2013.

[11] JEDEC, "Standard No. 79-3. DDR3 SDRAM STANDARD," 2010.

[12] JEDEC, "Standard No. 79-4. DDR4 SDRAM STANDARD," 2012.

[13] M. K. Jeong *et al.*, "Balancing DRAM locality and parallelism in shared memory CMP systems," in *HPCA*, 2012.

[14] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era," in *MICRO*, 2011.

[15] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010.

[16] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.

[17] Y. Kim *et al.*, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.

[18] D. Lee *et al.*, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.

[19] L. Liu *et al.*, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *PACT*, 2012.

[20] C. K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[21] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in SMT processors," in *ISPASS*, 2001.

[22] Micron, "Verilog: DDR3 SDRAM Verilog model."

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

[24] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX Security*, 2007.

[25] S. P. Muralidhara *et al.*, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *MICRO*, 2011.

[26] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO*, 2007.

[27] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *ISCA*, 2008.

[28] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *MICRO*, 2006.

[29] G. Nychis, C. Fallin, T. Moscibroda, and O. Mutlu, "On-chip networks from a networking perspective: Congestion and scalability in many-core interconnects," in *SIGCOMM*, 2012.

[30] S. Rixner *et al.*, "Memory access scheduling," in *ISCA*, 2000.

[31] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE CAL*, January 2011.

[32] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *ASPLOS*, 2000.

[33] L. Subramanian *et al.*, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *HPCA*, 2013.

[34] L. Tang *et al.*, "The impact of memory subsystem resource sharing on datacenter applications," in *ISCA*, 2011.

[35] H. Vandierendonck and A. Seznec, "Fairness metrics for multi-threaded processors," *IEEE CAL*, February 2011.

[36] Q. Zhu *et al.*, "A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing," in *3DIC*, 2013.

[37] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *ASPLOS*, 2010.

[38] W. K. Zuravleff and T. Robinson, "Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order," Patent 5630096, 1997.