Memory Access Scheduling Based on Dynamic Multilevel Priority in Shared DRAM Systems

DONGLIANG XIONG, KAI HUANG, XIAOWEN JIANG, and XIAOLANG YAN,
Zhejiang University

Interapplication interference at shared main memory severely degrades performance and increasing DRAM frequency calls for simple memory schedulers. Previous memory schedulers employ a per-application ranking scheme for high system performance or a per-group ranking scheme for low hardware cost, but few provide a balance. We propose DMPS, a memory scheduler based on dynamic multilevel priority. First, DMPS uses "memory occupancy" to measure interference quantitatively. Second, DMPS groups applications, favors latency-sensitive groups, and dynamically prioritizes applications by employing a per-level ranking scheme. The simulation results show that DMPS has 7.2% better system performance and 22% better fairness over FRFCFS at low hardware complexity and cost.

CCS Concepts: • **Hardware** → **Dynamic memory**; • **Computer systems organization** → *Multicore architectures*

Additional Key Words and Phrases: Memory access scheduling, hardware complexity, memory occupancy, dynamic multilevel priority, group

ACM Reference Format:
Dongliang Xiong, Kai Huang, Xiaowen Jiang, and Xiaolang Yan. 2016. Memory access scheduling based on dynamic multilevel priority in shared DRAM systems. ACM Trans. Archit. Code Optim. 13, 4, Article 42 (December 2016), 26 pages.
DOI: http://dx.doi.org/10.1145/3007647

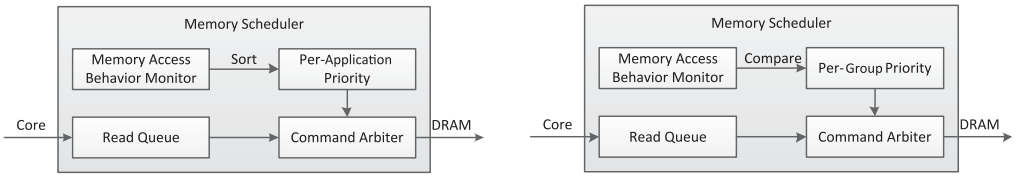
1. INTRODUCTION

In multicore systems, main memory is typically a critical shared resource due to its high access latency and limited memory bandwidth. Multiple applications executing concurrently contend heavily for the limited memory bandwidth and harmfully interfere with each other. Requests from an application can delay and even starve requests from other applications by causing additional row-buffer conflicts, bank conflicts, and address/data bus conflicts, and destroying intraapplication bank-level parallelism [Mutlu and Moscibroda 2008]. The interapplication interference, if not properly managed, can severely degrade overall system performance and fairness, especially at the trend of increasing numbers of cores on a chip and data intensity of applications [Moscibroda and Mutlu 2007].

To mitigate memory interference, smart resources, dumb resources, and integrating smart and dumb resources are three major approaches [Mutlu 2013; Mutlu et al. 2015].

This work is a new article, not an extension of a conference paper.
X. Yan is also at State Key Laboratory of ASIC & System, Fudan University, China.
This work is supported by Science Foundation of Zhejiang Province under grant LY14F020026.
Authors' addresses: D. Xiong, K. Huang (corresponding author), X. Jiang, and X. Yan, Institute of VLSI Design, Zhejiang University, China; emails: {xiongdl, huangk, jiangxw, yan}@vlsi.zju.edu.cn.

DOI: http://dx.doi.org/10.1145/3007647



(a) Per-application ranking mechanism (b) Per-group ranking mechanism

Fig. 1. Simplified architecture of application-aware memory scheduler.

The smart resources approach enables memory controllers [Mutlu and Moscibroda 2007, 2008; Moscibroda and Mutlu 2008; Kim et al. 2010a, 2010b; Ausavarungnirun et al. 2012; Jeong et al. 2012a; Chatterjee et al. 2014; Subramanian et al. 2014; Ebrahimi et al. 2011], interconnects [Das et al. 2009, 2010; Grot et al. 2009; Mishra et al. 2013], and caches [Khan et al. 2014; Qureshi and Patt 2006] to detect interapplication interference and prevent unfair applications slowdowns and performance degradation, whereas the dumb resources approach controls the resource allocation at the cores/sources or at the system software to mitigate unfair slowdowns and performance degradation [Ebrahimi et al. 2010; Das et al. 2013; Liu et al. 2012; Jeong et al. 2012b; Muralidhara et al. 2011; Xie et al. 2014]. Integrating the smart and dumb resources approach can enable more effective interference mitigation because these approaches are orthogonal [Muralidhara et al. 2011; Xie et al. 2014].

Application-aware memory access scheduling is a prevalent solution direction to mitigate interapplication interference, which focuses on the design of smart memory controllers. Typically, an application-aware memory scheduler consists of three parts: (1) monitoring applications' memory access behavior, used as the proxy for vulnerability to interference; (2) ranking applications based on memory access behavior such that vulnerable-to-interference applications have higher priority; and (3) selecting a ready command with the highest priority to issue. The hardware complexity and cost of a memory scheduler are determined by the granularity to rank applications and the complexity to monitor memory access behavior. Previously proposed application-aware memory schedulers are biased toward either system performance and fairness or hardware complexity and cost. The major difference is the ranking mechanism used to prioritize applications, as shown in Figure 1.

At one extreme, system performance can be achieved by applying the shortest-job-first principle. To maximize system performance, schedulers such as that shown in Figure 1(a) employ the per-application ranking mechanism and sort applications by memory intensity so that applications with lower memory intensity are strictly prioritized over others. Memory intensity has different meanings in different schedulers, such as max-total bank load in parallelism-aware batch scheduling (PARBS) [Mutlu and Moscibroda 2008], attained service in adaptive per-thread least-attained service memory scheduling (ATLAS) [Kim et al. 2010a], and misses per kilo instructions (MPKI) in thread cluster memory scheduling (TCM) [Kim et al. 2010b]. Fairness can be ensured by applying a higher priority rule, such as marked requests first in PARBS, age-over-threshold requests first in ATLAS, and insertion or random shuffle among bandwidth-sensitive applications in TCM. These schedulers can achieve high system performance and fairness, but they also incur very high hardware complexity and cost. Previous work [Subramanian et al. 2014] has demonstrated that the critical path latency of PARBS and TCM on a 24-core system has exceeded the minimum scheduling time t_{CCD} for DDR3-1333 and DDR4-3200. As the number of cores and DRAM frequency continue to increase, the situation will be worse. Therefore, keeping memory scheduler lean for quick scheduling actions becomes a critical consideration for design.

At the other extreme, schedulers such as that shown in Figure 1(b) employ the per-group ranking mechanism to keep hardware complexity low and coarsely classify applications into groups by simple comparison operations (i.e., nonblacklisted group blacklisted groups in the blacklisted memory scheduler (BLISS) [Subramanian et al. 2014, 2016]. To ensure fairness, BLISS prioritizes nonblacklisted applications and gives equal priority to applications in the same group. The hardware complexity and cost are very low, and the critical path latency of BLISS with a commercial 32nm standard cell library is lower than t_{CCD} for DDR4-3200. However, BLISS assigns the equal priority to all applications initially and blacklists interference-causing applications after they have caused enough interference. BLISS uses only one threshold to measure the ability to cause interference and cannot sense the variety of applications in memory access behavior. Therefore, BLISS sacrifices high system performance to pursue low hardware complexity and cost.

Our goal in this work is to design a simple memory scheduler that achieves high system performance and fairness. To this end, we propose DMPS, a new memory scheduler based on dynamic, multilevel priority. The key idea behind DMPS is to dynamically prioritize applications into multiple levels according to their caused interference. The design of DMPS is based on three observations.

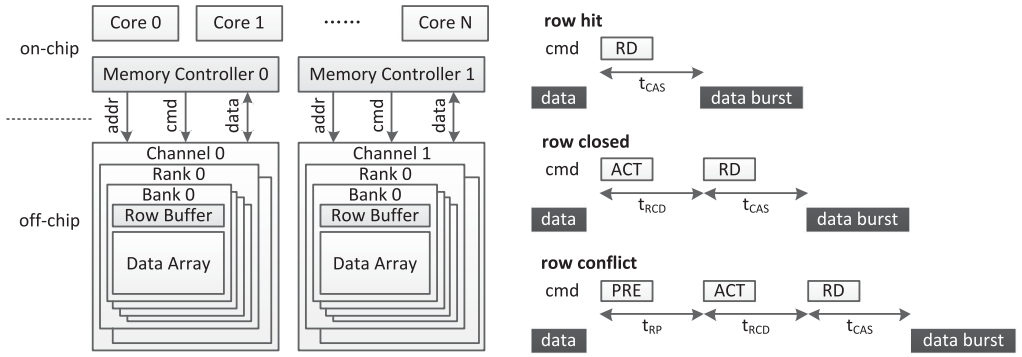
First, we use "memory occupancy" to measure the ability to cause memory interference quantitatively, which is defined as the ratio of the number of read requests served from an application to the average number of read requests served for each application. As a proxy to measure memory interference, memory occupancy can be implemented in hardware by simply adding several counters to store the number of read requests served. Memory occupancy takes the interference of both memory intensity and row-buffer locality into consideration. Applications with high memory occupancy are typical of applications with high memory intensity or high row-buffer locality, which can get more memory bandwidth and heavily cause interference.

Second, previous works [Mutlu and Moscibroda 2008; Kim et al. 2010a, 2010b; Zheng et al. 2008] have demonstrated that prioritizing latency-sensitive applications over bandwidth-sensitive applications can significantly improve system performance. Since latency-sensitive applications seldom generate memory requests, they have low memory occupancy and cause negligible interference to other applications even when prioritized. In DMPS, applications are also classified into latency- and bandwidth-sensitive groups based on their memory occupancy. To maximize system performance, latency-sensitive applications are prioritized over bandwidth-sensitive applications initially.

Third, prior works [Kim et al. 2010a, 2010b] have shown that unfairness problems come from the static priority among bandwidth-sensitive applications. Therefore, DMPS employs dynamic priority for applications to ensure fairness. Applications in the same group have equal priority initially. When the memory occupancy of an application exceeds predetermined thresholds, the application has attained too much memory bandwidth and its priority will be lowered by one level to ensure the fair share of memory bandwidth for other applications. Thus, the applications in the same group are treated differently. Although DMPS coarsely classifies applications into groups, fine-grain multilevel priority makes DMPS sensitive to applications' variety in memory access behavior. The multilevel priority also reduces hardware complexity greatly, and DMPS can be applied to the systems with more cores and faster DRAMs.

This article makes the following contributions:

—We propose the idea that classifying applications into multiple priority levels can achieve high system performance and fairness while incurring low hardware complexity and cost. A simple memory occupancy metric based on MPKC is used to measure the ability of an application to cause memory interference quantitatively.



(a) Logic hierarchy of memory system (b) Various latency of memory access

Fig. 2. Characteristics of a DRAM-based memory system.

Based on memory occupancy, applications are coarsely classified into latency- and bandwidth-sensitive groups, and latency-sensitive applications have higher initial priority. By comparing the number of requests served to predetermined thresholds, applications are prioritized dynamically into multiple levels.

—We compare DMPS to five previously proposed memory scheduling algorithms in terms of system performance and fairness on a 24-core and 4-channel system across 80 workloads. The results show that DMPS has 7.2% better system performance and 22.0% better fairness over FRFCFS, similar to PARBS.

—We implement the five previous schedulers and DMPS in RTL and compare their hardware complexity by comparing critical path latency and area. The results show that the hardware complexity of DMPS is much lower than TCM and PARBS, and a little higher than BLISS, FRFCFS-Cap.

The remainder of the article is organized as follows. A brief description of the DRAM-based memory system and previous memory scheduling algorithms is presented in Section 2. Section 3 describes the mechanism of DMPS in detail. Section 4 discusses the hardware implementation of DMPS. Section 5 shows details of our experimental setup, and Section 6 provides the simulation results and analysis. We offer our conclusions in Section 7.

2. BACKGROUND

2.1. DRAM-Based Memory System

Figure 2(a) shows a typical DRAM-based memory system in multicore systems, which is organized hierarchically as channels, ranks, and banks. Each channel has independent address, command, and data buses, and can achieve the highest degree of parallelism. Each channel consists of multiple ranks, and each rank consists of multiple banks operated in lockstep. Ranks or banks can operate in parallel, but the degree of parallelism is lower since all ranks and banks in a channel share the address, command, and data buses. The bank is a two-dimensional data array organized as rows and columns. The row buffer, an internal structure in each bank, acts as an interface between the data array and memory controller. Based on the state of the row buffer, memory access can be divided into three categories: an access to the row that is currently open in the row buffer is called a *row hit*, an access to the closed row buffer is called a *row closed*, and an access to the row different from the row that is currently open in the row buffer is called a *row conflict*. Figure 2(b) shows the access latency for these three categories. The

latency of a row hit, closed, and conflict is t_{CAS}, $t_{RCD}+t_{CAS}$, and $t_{RP}+t_{RCD}+t_{CAS}$,[1] respectively. Usually, a row hit is served approximately two times faster than a row closed and nearly three times faster than a row conflict. Thus, start-of-the-art memory schedulers prioritize row-hit requests over other requests to maximize DRAM throughput.

2.2. Application-Unaware Memory Schedulers

FRFCFS is a commonly used memory scheduling algorithm that prioritizes ready commands from (1) row-hit requests over other requests, and (2) if row-hit status is equal, older requests over younger requests. The goal of FRFCFS is to minimize the average service latency of memory accesses and maximize the DRAM throughput. FRFCFS can provide the best average performance in single-threaded systems [Rixner et al. 2000]. However, FRFCFS has poor system performance and fairness in multicore systems due to its application-unaware scheduling policies. FRFCFS favors requests from applications with high row-buffer locality or high memory intensity, delaying and even starving requests from others applications.

Since a large number of consecutive row hits can easily starve row-closed or row-conflict requests from other applications to the same bank, FRFCFS-Cap uses a counter per bank to monitor the number of younger column accesses served before older row accesses to the same bank. For banks in which the counter is below the parameter *Cap*, the FRFCFS policy is applied, whereas for banks in which the counter is over the parameter *Cap*, the FCFS policy is applied for fairness and the counter is reset to zero. Therefore, FRFCFS-Cap alleviates the problem of favoring requests from applications with high row-buffer locality. However, FRFCFS-Cap cannot solve the FCFS-inherent problem of penalizing requests from non–memory-intensive applications.

2.3. Application-Aware Memory Schedulers

In multicore systems, application-aware memory schedulers based on FRFCFS are developed in two directions: (1) to precisely quantify and control the impact of memory interference for predictable performance, such as the stall-time fair memory scheduler (STFM) [Mutlu and Moscibroda 2007], fairness via source throttle (FST) [Ebrahimi et al. 2010], per-thread cycle accounting [Eyerman and Eeckhout 2009], memory interference–induced slowdown estimation (MISE) [Subramanian et al. 2013], and the application slowdown model (ASM) [Subramanian et al. 2015]. In heterogeneous systems with hardware accelerators (HWAs), DASH [Usui et al. 2016] is designed to meet HWAs' deadlines and provide high CPU performance. In COTS-based multicore systems, Kim et al. [2014] bound the worst-case memory interference by combining request- and job-driven approaches. Kim et al. [2016] co-locate memory-intensive tasks on the same core with dedicated DRAM banks to reduce memory interference among tasks and develop a memory interference–aware task allocation algorithm to provide high task schedulability and to mitigate interference for high system performance and fairness, such as the adaptive history–based memory scheduler (AHB) [Hur and Lin 2004], the RL-based memory scheduler (RLMS) [Ipek et al. 2008], the fair thread-aware memory scheduler (FTAM) [Zhu et al. 2010], the request density–aware fair memory scheduler (RDAF) [Ikeda et al. 2012], the multiobjective reconfigurable self-optimizing memory scheduler (MORSE) [Mukundan and Martínez 2012], PARBS, TCM, ATLAS, BLISS, and so on.

2.3.1. Application-Based Marking. PARBS groups memory requests into batches and processes older batches over younger ones, providing fairness across applications and starvation freedom to requests. PARBS chooses the bank load to characterize applications'

[1] t_{CAS}, t_{RCD}, and t_{CAS} are timing parameters for DRAM shown later in Table III.

memory access behavior and forms a rank order among applications by using the max-total scheme. To maximize system performance, row-hit requests and requests from higher-ranked applications that are usually non–memory intensive are prioritized. The rank-based prioritization within batch preserves the intra-application bank-level parallelism. However, PARBS cannot preserve intraapplication bank-level parallelism among multiple memory controllers. Because the ranking of applications is computed at the beginning of a batch and the fine granularity of batches makes PARBS requiring significant coordination.

ATLAS selects the attained service from multiple memory controllers to characterize applications' memory access behavior and takes long-term behavior into account. The goal of ATLAS is to achieve high system performance and scalability to increasing numbers of cores and memory controllers. Applications are strictly prioritized with the least-attained service application receiving the highest priority. However, the increase in system performance comes at the cost of fairness. The most memory intensive applications incur high slowdowns when the less memory intensive applications are strictly prioritized over them.

TCM decouples the goal of system performance and fairness by separating applications into two clusters and optimizes for both by employing different scheduling policies in different clusters. To achieve high system performance, TCM prioritizes latency-sensitive applications over bandwidth-sensitive applications and enforces a strict priority in the latency-sensitive cluster, with the least memory intensive application receiving the highest priority. TCM introduces a new metric called *niceness* to characterize applications' memory access behavior. Applications with high row-buffer locality are less nice to others, whereas applications with high bank-level parallelism are nicer, so TCM needs to monitor applications' row-buffer locality and bank-level parallelism. To provide high fairness, TCM periodically shuffles the priority of applications in the bandwidth-sensitive cluster and ensures that the nicer applications are likely to receive the higher priority.

BLISS dynamically separates applications into two groups by simply counting the number of consecutive requests served from an application, so BLISS has low hardware complexity and cost. To mitigate memory interference, BLISS prioritizes the vulnerable-to-interference group over the interference-causing group. All applications are in the vulnerable-to-interference group initially. An application is blacklisted only when it has caused interference heavily, penalizing vulnerable-to-interference applications. BLISS cannot sense the variety of applications in memory access behavior, as only one threshold is used to measure interference. Thus, BLISS has low system performance as well.

FATM and RDAF use MPKC as a metric to measure memory interference. FATM maintains memory requests from different threads into different queues and schedules the request with highest priority among the headmost requests of each queue, so the requests from the same thread cannot be served out of order. FATM uses source thread, arriving time, and serving history to calculate the threads' priority. The priority calculation equation of thread i is $P_i = \alpha \times WT_i + \beta \times AST_i + \gamma$, where α, β, γ are the weights, WT_i is the waiting time of thread i, and AST_i is the accumulated serving times of thread i. Threads are ranked in a total order at any time. Requests from threads with high AST may be blocked for a long time. Since WT is 32-bit and AST is 16-bit in FATM, the value of threads' priority may be large, leading to high complexity, whereas DMPS dynamically prioritizes threads into multiple levels based on memory occupancy.

RDAF replaces MPKI of the original TCM by MPKC. For thread clustering, latency-sensitive threads are determined by sorting the number of read requests served in the last quantum. DMPS also uses the MPKC of threads in the last quantum, but latency-sensitive threads are determined by comparing their memory occupancy to

the parameter MOPL, avoiding complex sorting operations. For the fine-grain priority assignment, DMPS still uses MPKC, but RDAF uses read density, which is the number of waiting read requests in the read queue. The drawback of RDAF is similar to TCM: all threads are ranked in a total order. Threads with higher MPKC may be blocked by threads with lower MPKC for a long time.

2.3.2. Request-Based Marking. Criticality-aware schedulers in Ghose et al. [2013] are based on the criterion that the loads blocking the ROB head are critical. The commit block predictor (CBP) at the processor side tracks loads that have previously blocked the ROB and predicts the criticality of loads when they issue. Five CBP tables have been evaluated: Binary, BlockCount, LastStallTime, MaxStallTime, and TotalStallTime. The binary predictor requires an extra bit per request entry in each memory controller, whereas others require at least 14 bits per request entry, leading to complex logic for scheduling the request with the highest criticality. Ghose et al. [2013] have shown that the criticality-based predictor performs well in low contention, whereas DMPS is designed to target thread heterogeneity and high contention, like TCM and PARBS. Thus, the criticality-aware predictor is orthogonal to DMPS. In addition, DMPS can solve the starvation problem of requests in the criticality-aware schedulers.

AHB in Hur and Lin [2004] introduces two basic history-based arbiters: one to match the ratio of reads and writes, which tends to avoid bottlenecks within the reorder queues, and the other to minimize the expected latency of scheduled operations. Both goals can be partially achieved by periodically switching between two arbiters in a probabilistic manner. AHB can match the unknown command pattern by implementing multiple history-based arbiters optimized for a different command pattern and periodically selecting the most appropriate one. AHB is simple and can improve DRAM throughput of DDR2 in single-threaded systems, but it cannot solve the memory interference problem in multithreaded systems. In a high-speed DDR3/4 environment, the command pattern arbiter in AHB may be ineffective due to a write drain policy.

RLMS [Ipek et al. 2008] and MORSE [Mukundan and Martínez 2012] apply the Q-learning algorithm in reinforcement learning and can learn to optimize its scheduling policy on the fly to maximize long-term performance. However, they have very high hardware overhead due to large number of Q-values.

3. MECHANISM

3.1. Simple Memory Occupancy to Measure Interference

We use memory occupancy to measure the ability to cause memory interference. The mathematic definition of memory occupancy is shown in Equation (1). The numerator is the number of read requests served from an application in the last quantum, and the denominator is the average number of read requests served for each application in the last quantum. Therefore, the memory occupancy of an application is determined by the application itself and other applications concurrently executing. Thus,

$$MO_i = \frac{ReqCnt_i}{\sum_{j=1}^{N_{core}} ReqCnt_j / N_{core}} = \frac{N_{core} \times MPKC_i}{\sum_{j=1}^{N_{core}} MPKC_j}, \quad (1)$$

where MO_i is the memory occupancy of application i, $ReqCnt_i$ is the number of read requests served from application i in a quantum, N_{core} is the number of currently executing applications, and $MPKC_i$ is the misses per kilo cycles of application i in a quantum.

Essentially, the memory occupancy is based on MPKC. As a proxy for vulnerability to interference, MPKC is simple to implement in hardware and directly reflects the ability of an application to occupy memory bandwidth when running with other

applications. MPKC is the result of MPKI, row-buffer locality, bank-level parallelism, or other processor-side behavior, whereas MPKI only shows the memory intensity of applications. In the schedulers using MPKI, other metrics, such as row-buffer locality, will be used as well. To determine the ability of applications to cause interference, a function of these metrics is carefully designed. In FATM, $P_i = \alpha \times WT_i + \beta \times AST_i + \gamma$. In TCM, bandwidth usage is used in thread clustering, and MPKI, row-buffer locality, and bank-level parallelism are sorted to determine the priority of applications. BLISS monitors the maximum number of consecutive read requests served from an application and blacklists the application when the maximum number is over the blacklisting threshold, so it cannot control the number of read requests served before interference-causing applications are blacklisted. However, MPKC uses the total number of read requests served from an application and has no such problem. Therefore, MPKC can provide the control of bandwidth usage and is more suitable than MPKI for schedulers pursuing low complexity.

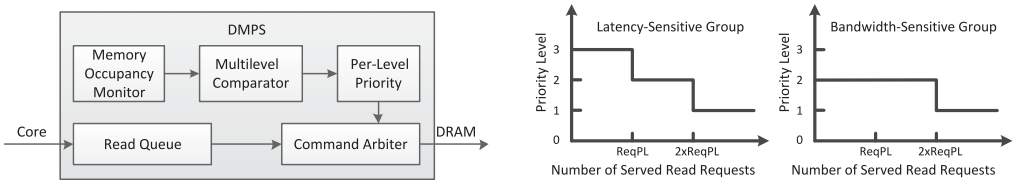
From the view of bandwidth usage, applications with high MPKC are more interference causing. To alleviate memory interference, DMPS dynamically assigns applications into multiple priority levels. *ReqPL* is the average MPKC per level in an epoch, determined by the long-term MPKC *QReqCnt* of all applications in a quantum and the configurable parameter *MOPL* below. *EReqCnt* is the short-term MPKC in an epoch. The priority level of an application is determined by comparing its *EReqCnt* with predetermined thresholds, which are multiples of *ReqPL*.

3.2. Coarse-Grain Synchronization to Group Applications

Algorithm 1 shows the process of synchronization in detail. The synchronization is performed every quantum, including the accumulation of $QReqCnt_{i,j}$, the grouping of applications, and the update of per-level threshold *ReqPL*. First, $QReqCnt_{i,j}$, the number of read requests served from application i in channel j, is sent from multiple memory controllers to the metacontroller at the end of every quantum. $ReqCnt_i$ is the sum of $QReqCnt_{i,j}$ across all channels, and *TotalReqCnt* is the sum of $ReqCnt_i$ across all applications.

Second, *MOPL* in Algorithm 1 is the value of memory occupancy per priority level, and *PLNUM* in Algorithm 2 is the number of priority levels. Both *MOPL* and *PLNUM* are configured by users. To separate applications into multiple levels effectively, the product of *MOPL* and *PLNUM* is about one. In DMPS, applications with memory occupancy larger than *MOPL* are treated as bandwidth sensitive, and others are treated as latency sensitive. Since it is complex and costly to calculate memory occupancy by division, *GroupTh* is used to classify applications instead of *MOPL*, where *GroupTh* is the corresponding value in terms of the number of read requests served. Applications with *ReqCnt* larger than *GroupTh* are labeled as bandwidth sensitive in current quantum, and the others are labeled as latency sensitive. The group state of applications in the next quantum, *NxtGroup*, is predicted by their past two classifications, *PreGroup* and *CurGroup*. An application is predicted as bandwidth sensitive if it was classified as bandwidth sensitive in past two quanta. Otherwise, the application is predicted as latency sensitive. Compared to using the last classification directly, the two-bit predictor taking into account past two classifications is more stable and accurate and gives latency-sensitive applications more opportunity to stay in the latency-sensitive group.

Third, *ReqPL* is determined by *MOPL* and *TotalReqCnt* in the current quantum, as shown in Algorithm 1. *ReqPL* is used as the per-level threshold to update dynamic priority of applications in the next quantum. When lots of read requests are served in the current quantum, memory interference is heavy and the value of *ReqPL* is updated to high. A high *ReqPL* gives applications with lower MPKC more opportunity to stay



(a) Simplified architecture (b) Mapping function

Fig. 3. Fine-grain dynamic multilevel priority.

ALGORITHM 1: Grouping Algorithm

Input: $QReqCnt_{i,j}$ (number of read requests served from application i in channel j) and $MOPL$ (value of memory occupancy per priority level)
Output: $NxtGroup[N_{core}]$ (predicted group state in the next quantum) and $ReqPL$ (number of read requests served per priority level per application per channel per epoch)

// Execute once at the end of every quantum
$TotalReqCnt = 0$;
for $i = 0$ *to* $N_{core} - 1$ **do**
  $ReqCnt_i = 0$;
  for $j = 0$ *to* $N_{channel} - 1$ **do**
    $ReqCnt_i += QReqCnt_{i,j}$;
  end
  $TotalReqCnt += ReqCnt_i$;
end
$GroupTh = TotalReqCnt \times MOPL \div N_{core}$;
for $i = 0$ *to* $N_{core} - 1$ **do**
  if $ReqCnt_i < GroupTh$ **then**
    $CurGroup[i] = 0$; // Latency-Sensitive
  else
    $CurGroup[i] = 1$; // Bandwidth-Sensitive
  end
  $NxtGroup[i] = PreGroup[i] \cap CurGroup[i]$; // Two-Bit Predictor
  $PreGroup[i] = CurGroup[i]$; // $PreGroup[i]$ is initialized to 0
end
$ReqPL = TotalReqCnt \times MOPL \times Epoch \div Quantum \div N_{core} \div N_{channel}$;

at high priority, alleviating memory interference. Finally, the metacontroller sends the predicted group state of applications *NxtGroup* and the per-level threshold *ReqPL* to each memory controller. The length of the quantum in DMPS is set to one million cycles like ATLAS and TCM, which is short enough to detect phase changes in applications' memory access behavior and long enough to minimize the communication overhead between the metacontroller and multiple memory controllers.

3.3. Fine-Grain Dynamic Multilevel Priority

Figure 3(a) shows the simplified architecture of DMPS. In the memory occupancy monitor, each application needs two counters to store the number of read requests served: a long-time counter *QReqCnt* for a quantum and a short-time counter *EReqCnt* for an epoch. The long-time counter *QReqCnt* is sent to the metacontroller for synchronization at the end of every quantum, and the short-time counter *EReqCnt* is sent to a multilevel comparator every cycle. The multilevel comparator performs the mapping function from the number of read requests served to multiple priority levels. Figure 3(b) shows the mapping function of DMPS with three priority levels, and the higher value of priority levels corresponds to the higher priority. To determine the priority level of

an application in an epoch, the short-time counter *EReqCnt* needs to compare to *ReqPL* and $2 \times ReqPL$, where *ReqPL* is the number of read requests served per priority level. When *EReqCnt* is smaller than *ReqPL*, the application is in the initial priority level. When *EReqCnt* is not smaller than $2 \times ReqPL$, the priority level of the application is one. In other situations, the priority level of the application is two. Algorithm 2 shows the process of fine-grain dynamic multilevel priority in detail.

The mapping function has two features. First, most applications with low MPKC are prioritized over applications with high MPKC to provide high system performance. Since applications with low MPKC seldom generate memory requests, they tend to have a high potential for making fast progress. In a quantum, applications in the latency-sensitive group are prioritized over that in the bandwidth-sensitive group initially. In an epoch, applications with small *EReqCnt* can be treated as latency sensitive temporarily, and most of them are prioritized over other applications in the same group. Second, applications in the same group have the same priority level initially. When the number of read requests served from an application is over predetermined thresholds, the priority of the application is lowered by one level to alleviate memory interference. Thus, most applications with similar *EReqCnt* are in the same priority level. The applications in low priority levels are usually bandwidth sensitive, and the same priority level can ensure the fair share of memory bandwidth. Therefore, the mapping function makes DMPS progress all applications at a relatively even and fast pace, providing high system performance and fairness.

The length of an epoch is kilos of cycles, which is far less than the length of a quantum. The major function of a fine-grain epoch is to keep the value of *ReqPL* small and ensure the effectiveness of dynamic priority. When *ReqPL* is too large, it needs a long time to lower applications by one level and the dynamic priority may be weakened to the static priority, leading to high unfairness and causing starvation. Another function of a fine-grain epoch is to avoid starvation. If a request from an application at a low priority level is blocked in the current epoch, the dynamic priority of the application is reset to the initial priority level at the beginning of the next epoch and the request will be prioritized among the requests with same priority levels due to older age. Thus, requests with low priority are serviced in a timely manner.

3.4. Summary: DMPS Prioritization Rules

Algorithm 3 summarizes how DMPS schedules memory requests from applications. For two memory requests to the same bank, a request from the application at a higher priority level is prioritized. When two applications are at the same priority level, then a row-hit request is prioritized. If all are equal, the older request is prioritized.

3.5. Support for Software Control

DMPS can support weights assigned by the system software, and therefore applications with high weights are prioritized during scheduling. TCM has shown that blindly prioritizing applications with large weights without considering their memory access behavior can severely degrade system performance and fairness. Thus, DMPS also honors software weights in the scope of the group. Latency-sensitive applications with small weights are prioritized over bandwidth-sensitive applications with large weights. To achieve this, the metacontroller does not care about software weights when grouping. DMPS can incorporate software weights by assigning different per-level thresholds to different applications. The per-level threshold for an application is the product of a per-level threshold from the metacontroller and its software weight. A high per-level threshold means that the application can serve more read requests before its priority

ALGORITHM 2: Dynamic Multilevel Priority

Input: $NxtGroup[N_{core}]$ (predicted group state in the current quantum), $PLNUM$ (number of priority levels), and $ReqPL$ (number of served read requests per priority level in an epoch)
Output: $QReqCnt[N_{core}]$ (number of served read requests in the current quantum)

```
// Initialization at the beginning of every quantum
QReqCnt[N_core] = {0};
EReqCnt[N_core] = {0}; // Number of served read requests in an epoch
IniPri[N_core] = PLNUM − NxtGroup[N_core]; // Initial priority
// During quantum
repeat
    // Memory Occupancy Monitor
    if A read request from application i is served then
        QReqCnt[i] + +;
        EReqCnt[i] + +;
    end
    // Multi-Level Comparator
    for i = 0 to N_core − 1 do
        if EReqCnt[i] < ReqPL then
            DynPri[i] = IniPri[i]; // Dynamic priority
        end
        else if EReqCnt[i] >= (PLNUM − 1) × ReqPL then
            DynPri[i] = 1;
        end
        else
            DynPri[i] = PLNUM − ⌊EReqCnt/ReqPL⌋;
        end
    end
    // Cleared every epoch
    if Epoch is expired then
        for i = 0 to N_core − 1 do
            EReqCnt[i] = 0;
            DynPri[i] = IniPri[i];
        end
    end
until End of quantum;
```

ALGORITHM 3: Prioritization Rules of DMPS

(1) **Highest level first:** Requests from applications at higher priority levels are prioritized.
   —The priority level of latency-sensitive applications is higher than that of bandwidth-sensitive applications by one level initially.
   —The priority of an application is lowered by one level when the number of served read requests is over predetermined thresholds.
(2) **Row-hit first:** Row-hit requests are prioritized over row-closed and row-conflict requests.
(3) **Oldest first:** Older requests are prioritized over younger requests.

level is lowered.

$$ReqPL_i = ReqPL \times SWWeight_i \tag{2}$$

4. IMPLEMENTATION AND HARDWARE COST

DMPS requires additional logic and storage over FR-FCFS to (1) monitor memory occupancy and (2) support dynamic multilevel priority. Table I shows the additional

Table I. Additional Hardware Cost Per Memory Controller for DMPS

Storage	Function	Size (bits)
IniPri	Initial priority per application in a quantum	N_{core}
DynPri	Dynamic priority per application in an epoch	$2N_{core}$
ReqPL	Number of served read requests per priority level in an epoch	9
EReqCnt	Number of served read requests per application in an epoch	$9N_{core}$
QReqCnt	Number of served read requests per application in a quantum	$16N_{core}$
Logic	**Function**	**Number**
9-bit adder	Increase EReqCnt by one when a request from this application is served	N_{core}
16-bit adder	Increase QReqCnt by one when a request from this application is served	N_{core}
9-bit comparator	Update dynamic priority for this application	$2N_{core}$

storage and logic cost per memory controller on a 24-core and 4-channel system. To monitor memory occupancy, each application has two counters to record the number of read requests served: a short-time counter for an epoch and a long-time counter for a quantum. The update of each counter needs an adder with the same width. In the worst case when every request is a row hit, the maximum available value for two counters is the result of dividing the length of the epoch and quantum by the minimum scheduling distance t_{CCD}. Thus, the width of two counters is 9 bits and 16 bits, respectively. To support dynamic multilevel priority, DMPS needs a 1-bit register per application to store the initial priority, a 2-bit register per application to store the dynamic priority, and a 9-bit register to store the threshold *ReqPL*. The dynamic priority of an application is updated every cycle by comparing its short-time counter to the thresholds *ReqPL* and $2 \times ReqPL$, so two 9-bit comparators are needed per application.

The metacontroller for DMPS needs three 18-bit adders per application and twenty-three 23-bit adders to calculate the total number of served requests in the last quantum. The group threshold *GroupTh* and the per-level threshold *ReqPL* for dynamic priority needs complex calculation, such as multiplication and division. However, the complex calculation can be replaced by simple right shift operations at the cost of accuracy. The hysteresis logic is simple: only a 1-bit register and an AND operation per application. We quantitatively evaluate the critical path latency and area for the metacontroller and each memory controller, and the result is shown in Section 6.1.

5. METHODOLOGY AND METRIC

5.1. System Configuration

We evaluate DMPS and five previous schedulers by using the memory system simulator USIMM [Chatterjee et al. 2012] for the Memory Scheduling Championship (MSC). Table II shows the configuration of the processors and memory system. In the baseline system, DRAM main memory is the only shared resource for isolating the memory interference. The USIMM simulator can model a DRAM-based memory system in detail with all required timing parameters. The DRAM device used is DDR3 4Gbx4 running at 1066Mbps, and its timing parameters are shown in Table III. The memory controller has a centralized read queue and write queue to store read and write requests, respectively. We use the simple write drain policy and refresh policy inside USIMM directly. The write drain mode is enabled when the length of the write queue is over the high watermark or the read queue is empty; otherwise, the read drain mode is enabled. When the length of the write queue is over the high watermark, the memory controller handles writes in succession until the length of the write queue is below the low watermark. When the refresh interval is expired, both read and write requests are

Table II. Configuration for Baseline System

Processor Configuration	
Number of cores	24
Frequency	2.132GHz
Pipeline depth	10
Fetch/retire width	4/4
ROB size	160
Last-level cache (private)	512KB per core
Block size	64 bytes
Memory Controller Configuration	
Read/write queue	128/128 entries
Write queue high/low watermark	80/40
Address mapping policy	Row:Rank:Bank:Channel:Column:Block
Scheduling policy	FRFCFS
Page policy	Open page
DRAM Configuration: DDR3 4Gbx4 devices	
Clock frequency	533MHz
Number of channels	4
Ranks per channel	1
Banks per rank	8
Page size	16KB

Table III. Timing Parameters for a 4Gbx4 1066Mbps DDR3 DRAM Device

Parameter	DRAM Cycles	Parameter	DRAM Cycles
t_{CAS} (t_{CL})	8	t_{RTP}	4
t_{RCD}	8	t_{CWD}	6
t_{RP}	8	t_{RRD}	4
t_{RAS}	20	t_{FAW}	20
t_{RC}	28	t_{RTRS}	2
t_{CCD}	4	t_{RFC}	139
t_{WR}	8	t_{REFI}	4,160
t_{WTR}	4	t_{Burst}	4

blocked and refresh operations are forced to perform. The page policy for row-buffer management is open page by default, where a bank is only precharged if there are pending references to other rows in the bank and there are no pending references to the active row. For all evaluated schedulers, their differences are the scheduling policies applied on read requests.

5.2. Workload Configuration

We drive the USIMM simulator by the memory traces of SPEC CPU2006 [Henning 2006]. These memory traces are provided in Ramulator [Kim et al. 2015] and are representative phases chosen by PinPoints [Patil et al. 2004]. We classify the benchmarks into four categories based on their memory intensity (1: high, 0: low) and row-buffer locality (1: high, 0: low). Benchmarks with a last-level cache MPKI greater than 3 are labeled as high memory intensity, and others are labeled as low memory intensity. Benchmarks with a row-buffer hit rate greater than 0.8 are labeled as high row buffer locality, and others are labeled as low row buffer locality. Table IV shows the characteristics and categories of individual benchmarks when they run alone on the baseline system. The memory intensity of a workload is defined as the percentage of benchmarks with high memory intensity. We formed the 80 multiprogrammed workloads by varying

Table IV. Memory Access Behavior of Benchmarks

Benchmark	MPKI	RBHR	Category	Benchmark	MPKI	RBHR	Category
444.namd	0.107	0.907	01	437.leslie3d	7.238	0.912	11
447.dealII	0.116	0.883	01	482.sphinx3	13.214	0.913	11
481.wrf	0.139	0.873	01	433.milc	13.811	0.760	10
435.gromacs	0.195	0.897	01	483.xalancbmk	18.536	0.832	11
403.gcc	0.224	0.713	00	471.omnetpp	19.767	0.812	11
458.sjeng	0.359	0.089	00	462.libquantum	25.000	0.996	11
445.gobmk	0.541	0.587	00	434.zeusmp	25.286	0.800	10
464.h264ref	1.255	0.827	01	470.lbm	26.689	0.706	10
456.hmmer	2.817	0.929	01	459.GemsFDTD	26.830	0.688	10
436.cactusADM	4.714	0.232	10	450.soplex	27.368	0.884	11
473.astar	5.577	0.779	10	429.mcf	72.898	0.017	10

RBHR, row-buffer hit rate. The category is determined based on MPKI (1: high, 0: low) and RBHR (1: high, 0: low).

memory intensity from 25%, 50%, 75%, to 100% and selecting benchmarks randomly. Each workload is simulated for 100 million representative cycles.

5.3. Evaluation Metrics

We evaluate six schedulers by three commonly used metrics for multiprogrammed environments: weighted speedup [Snavely and Tullsen 2000] to measure system performance, maximum slowdown to measure fairness, and harmonic speedup [Luo et al. 2001] for a balance of fairness and system performance. Thus, we have

$$Weighted\ Speedup = \sum_{i=1}^{n} \frac{IPC_i^{shared}}{IPC_i^{alone}}, \tag{3}$$

$$Harmonic\ Speedup = \frac{N}{\sum_{i=1}^{n} \frac{IPC_i^{alone}}{IPC_i^{shared}}}, \tag{4}$$

$$Maximum\ Slowdown = max_i \frac{IPC_i^{alone}}{IPC_i^{shared}}, \tag{5}$$

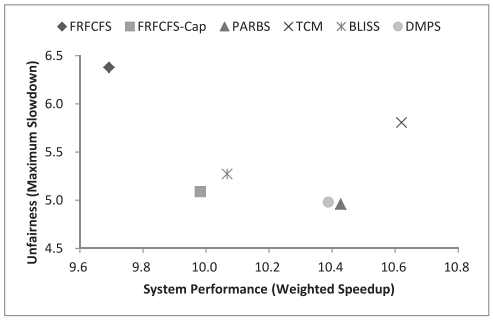
where IPC_i^{alone} is instructions per cycle for application i when running alone, and IPC_i^{shared} is instructions per cycle for application i when running with other applications.

5.4. Parameters for Evaluated Schedulers

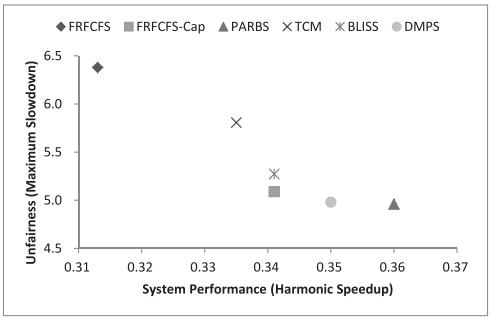
FRFCFS has no parameters. We use a cap of 4 for FRFCFS-Cap and a *Marking-Cap* of 4 for PARBS. For TCM, we set *ClusterThresh* to 4/24, *ShuffleInterval* to 800 cycles, *Quantum* to 1 million cycles, and *ShuffleAlgoThresh* to 0.1. For BLISS, we set 2 as the *BlacklistingThreshold* and 1,000 cycles for *Clearing Interval*. For the proposed DMPS, we set *MOPL* to 0.3, *PLNUM* to 3, *epoch* to 5,000 cycles, and *quantum* to 1 million cycles. These parameters provide a good balance between system performance and fairness on our system configuration.

6. EVALUATION RESULTS

We evaluate five prior memory schedulers (FRFCFS, FRFCFS-Cap, PARBS, TCM, and BLISS) and DMPS in the baseline system. The geometric mean across the default 80



(a) Weighted speedup versus maximum speedup



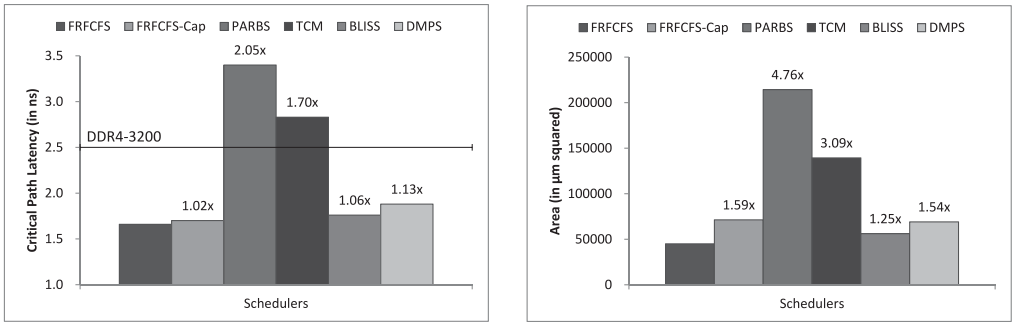
(b) Harmonic speedup versus maximum speedup

Fig. 4. Pareto plots of system performance and fairness.

workloads of different memory intensity is used as the final result. Figure 4 shows two Pareto plots of system performance (weighted speedup and harmonic speedup) and unfairness (maximum slowdown). The lower right part of the two plots corresponds to a better fairness and higher system performance. We can draw three conclusions.

First, DMPS outperforms application-unaware schedulers FRFCFS and FRFCFS-Cap and per-group ranking scheduler BLISS in both system performance and fairness. For system performance, DMPS improves weighted speedup and harmonic speedup by 7.2% and 11.8% compared to FRFCFS, 4.1% and 2.6% compared to FRFCFS-Cap, and 3.2% and 2.6% compared to BLISS. For fairness, DMPS reduces maximum slowdown by 22% compared to FRFCFS, 2.1% compared to FRFCFS-Cap, and 5.6% compared to BLISS. Application-unaware schedulers are prone to starvation, degrading system performance significantly. To mitigate interapplication interference, the per-group ranking scheduler BLISS blacklists an application when the amount of consecutive requests served is over the blacklisting threshold. The simple group scheme of BLISS is dynamic—that is, the group state of an application can change in a clearing interval. However, BLISS cannot mitigate the memory interference caused by interference-causing applications before they are blacklisted. In addition, BLISS has the FCFS-inherent problem of penalizing applications with less memory intensity in the blacklisted group. In DMPS, the group state of an application is unchanged and latency-sensitive applications are prioritized over bandwidth-sensitive applications initially, avoiding the interference among applications from different groups. DMPS dynamically prioritizes bandwidth-sensitive applications into multiple levels and assigns lower priority to applications with higher memory occupancy, ensuring the fair share of limited bandwidth. In other words, DMPS can provide better system performance and fairness than FRFCFS, FRFCFS-Cap, and BLISS.

Second, compared to the per-application ranking scheduler TCM, DMPS improves harmonic speedup and fairness by 4.5% and 14.3%, respectively, and degrades weighted speedup by 2.2%. Although TCM achieves the best weighted speedup among simulated schedulers, TCM has very high hardware complexity and cost and poor fairness. To improve system performance, TCM separates applications into clusters by MPKI and favors the applications with low MPKI. Similarly, DMPS also classifies applications into two groups by MPKC, and applications with low MPKC tend to be prioritized over applications with high MPKC. To provide high fairness, TCM shuffles bandwidth-sensitive applications by niceness, which is related to row-buffer locality and bank-level parallelism. In each shuffle, TCM ranks applications in a static and total order, unfairly penalizing the applications at the bottom of ranking stack, whereas DMPS dynamically prioritizes applications into multiple priority levels, and applications with similar



(a) Critical path (b) Area

Fig. 5. Hardware complexity: DMPS versus previous schedulers.

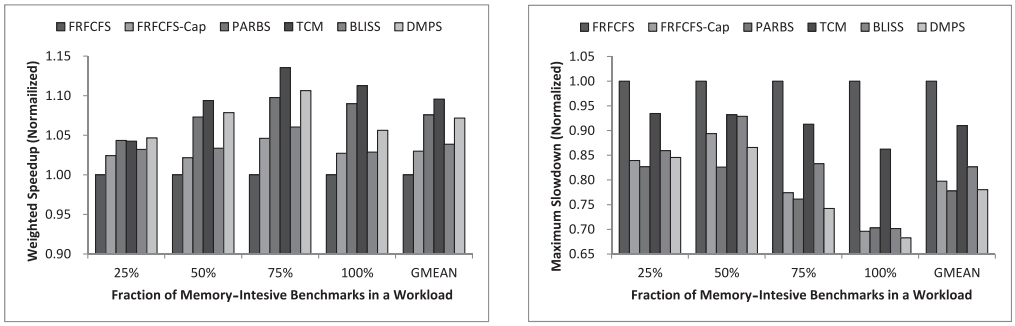
MPKC have the same priority, ensuring the fair share of bandwidth. Due to per-thread ranking, TCM has higher hardware complexity than DMPS. Although DMPS degrades system performance slightly, DMPS is much simpler and fairer than TCM.

Third, DMPS has similar system performance and fairness to PARBS and decreases by 0.4% in weighted speedup, 2.8% in harmonic speedup, and 0.04% in fairness. PARBS provides the best fairness and harmonic speedup among simulated schedulers. This is because the batching policy in a fine granularity is quite effective in providing fairness across applications and avoiding starvation of requests. Within each batch, PARBS monitors the maximum and total bank load per application, ranks applications in a total order by the max-total scheme to improve intra-application bank-level parallelism, and prioritizes applications with low memory intensity to improve system performance. The batching policy and per-application ranking mechanism make the hardware complexity of PARBS even higher than that of TCM. Prior work [Subramanian et al. 2014] has shown than the critical path latency of PARBS cannot meet the minimum scheduling time t_{CCD} for DDR3-800. The storage cost of PARBS is lower than the storage cost of TCM, yet it is still higher than the storage cost of DMPS, as shown in the next section. Thus, at the cost of slight degradation in system performance and fairness, DMPS has much lower hardware complexity and cost than PARBS.

6.1. Hardware Complexity

We write the register transfer level (RTL) versions of all evaluated schedulers in Verilog and use Design Compiler to synthesize the RTL implementations. The standard cell library is TSMC45G. The lowest value of critical path latency and corresponding area for each memory controller are shown in Figure 5.

First, PARBS and TCM have 1.05 and 0.70 times longer critical path latency and 3.76 and 2.09 times larger area than FRFCFS, respectively. This is due to their complex per-application ranking mechanism. On one hand, the per-application ranking increases the width of comparators to select the request with the highest priority by $log_2 NumOfPriority$ bits: 6 bits for PARBS and 5 bits for TCM. On the other hand, the logic to generate per-application ranking is complex and costly. PARBS needs to mark the requests per bank per application and sorts the applications by the max-total scheme. TCM needs to shuffle the priority of bandwidth-sensitive applications periodically and monitors complex memory access behavior (e.g., row-buffer locality, bank-level parallelism, bandwidth usage). As shown in Figure 5, the critical path latency of both PARBS and TCM is longer than 2.5ns, which is the worst-case scheduling time t_{CCD} for DDR4-3200. As a result, PARBS and TCM cannot be applied directly to memory systems with DDR4-3200.



(a) System performance (b) Unfairness

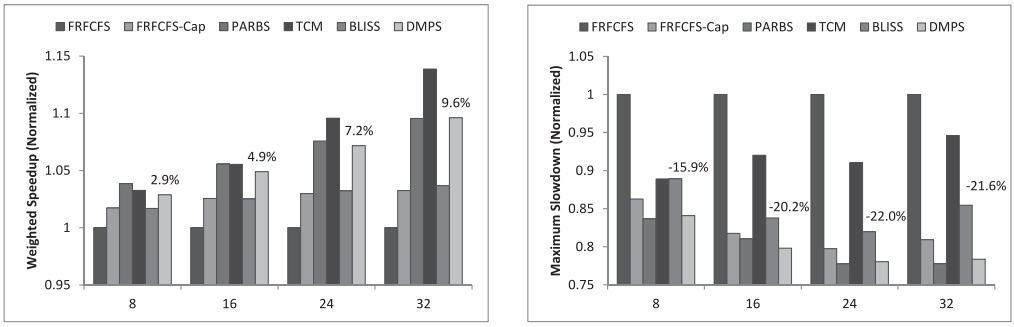
Fig. 6. Sensitivity to different workload intensity.

Second, in terms of critical path latency and area, the increase of FRFCFS-Cap, BLISS, and DMPS over FRFCFS is 1.02, 1.06, and 1.13 times and 1.59, 1.25, and 1.54 times, respectively. These schedulers are simple and can meet the worst-case scheduling timing t_{CCD} for DDR4-3200. FRFCFS-Cap implements two arbitration rules, FRFCFS and FCFS, per bank and selects one rule based on the cap state of each bank, so FRFCFS-Cap has a higher area. The sources of additional latency for BLISS and DMPS are the wider tie-breaking age comparator (major part) and the logic to update dynamic priority of applications. DMPS needs two bits to represent the dynamic priority of applications. The additional area of DMPS over FRFCFS comes from the more complex logic to select the request with the highest priority (major part) and the additional storage cost.

Finally, the critical path latency and area of the metacontroller is 1.28ns and $34{,}121\mu m^2$. The complex calculation of *GroupTh* and *MOPL* is replaced by a simple right shift operation at the cost of little accuracy. Hence, the hardware complexity of DMPS is much lower than per-application ranking schedulers (e.g., TCM, PARBS) and a little higher than the per-group scheduler (e.g., BLISS), and DMPS is suitable for faster DRAM technologies of the future.

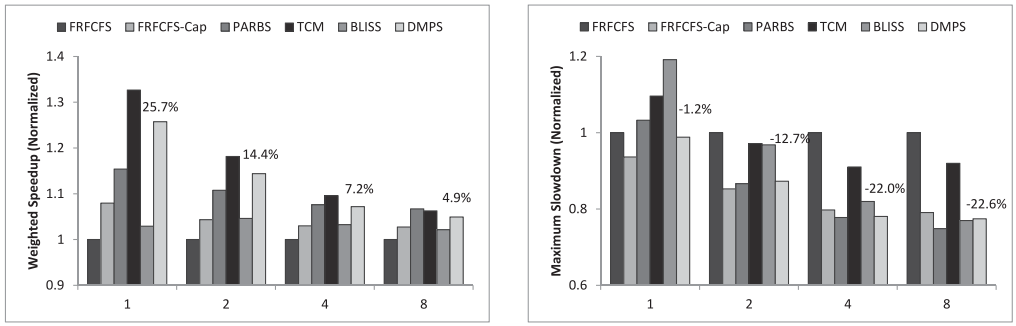
6.2. Sensitivity to Workload Memory Intensity

The simulated 80 workloads consist of four memory intensity categories representing different levels of contention. The memory intensity category is determined by the percentage of memory-intensive benchmarks in a workload. Figure 6 shows the weighted speedup and maximum slowdown of six schedulers for each category. First, DMPS achieves better system performance than FRFCFS, FRFCFS-Cap, and BLISS across all categories. In the 75% workload intensity category, DMPS improves system performance by 10.6% compared to FRFCFS, by 5.8% compared to FRFCFS-Cap, and by 4.3% compared to BLISS. Second, the system performance of DMPS is nearly same as that of PARBS, except DMPS has 3.1% lower system performance than PARBS in the 100% workload intensity category. Compared to TCM, DMPS benefits less in system performance when the workload memory intensity increases: 2.6% in the 75% category and 5.1% in the 100% category. The reason for low performance in the 100% memory intensity category is that all applications have high memory intensity and nearly no applications are treated as latency sensitive by memory occupancy. Third, fairness benefits of DMPS increase significantly with the workload memory intensity. This is due to the dynamic multilevel priority, and no application is unfairly slowed down. DMPS achieves the best fairness in the 75% and 100% workload intensity categories, where applications have a high demand for memory bandwidth and a fair



(a) System performance (b) Unfairness

Fig. 7. Sensitivity to different cores.



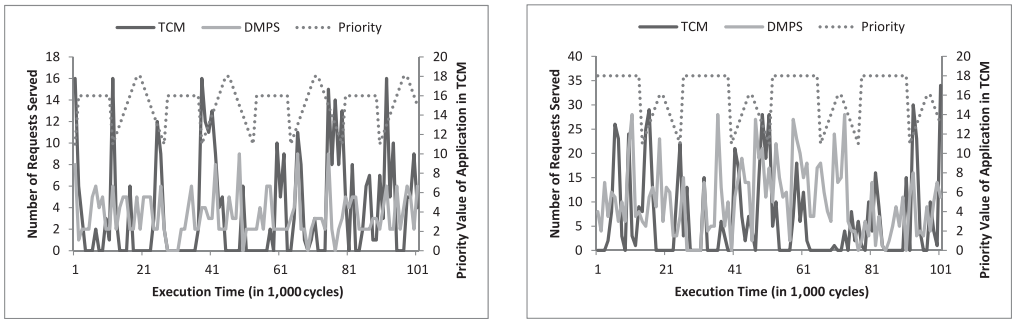
(a) System performance (b) Unfairness

Fig. 8. Sensitivity to different channels.

share of memory bandwidth is more important than high performance. In the 100% workload intensity category, DMPS reduces maximum slowdown by 31.7% compared to FRFCFS, 2.9% compared to PARBS, 21.7% compared to TCM, and 2.7% compared to BLISS. Consequently, DMPS can adapt to the workload intensity and achieve high system performance and fairness in different levels of contention.

6.3. Sensitivity to Cores and Channels

Figure 7 shows the system performance and fairness by varying the core counts on the baseline system. Figure 8 shows the system performance and fairness by varying the channel counts on the baseline system. The system performance and fairness benefits of DMPS over FRFCFS are displayed over the bars. First, system performance benefits of DMPS over FRFCFS increase stably as the core counts increase, whereas fairness benefits of DMPS over FRFCFS increase slightly after 16 cores. On a 32-core and 4-channel system, DMPS improves system performance and fairness by 9.6% and 21.6% compared to FRFCFS. Second, system performance benefits of DMPS over FRFCFS decrease dramatically with the channel counts, whereas the fairness benefits of DMPS over FRFCFS increase significantly. The fairness benefit is low at very low channel counts and very high core counts, only 1.2% on the 24-core and 1-channel system, as memory bandwidth is highly saturated in these systems. Third, system performance and fairness benefits of DMPS over FRFCFS-Cap, PARBS, and BLISS are similar to that of DMPS over FRFCFS. Compared to PARBS, DMPS improves system performance and fairness by 3.3% and 0.8% on the 24-core and 2-channel system and



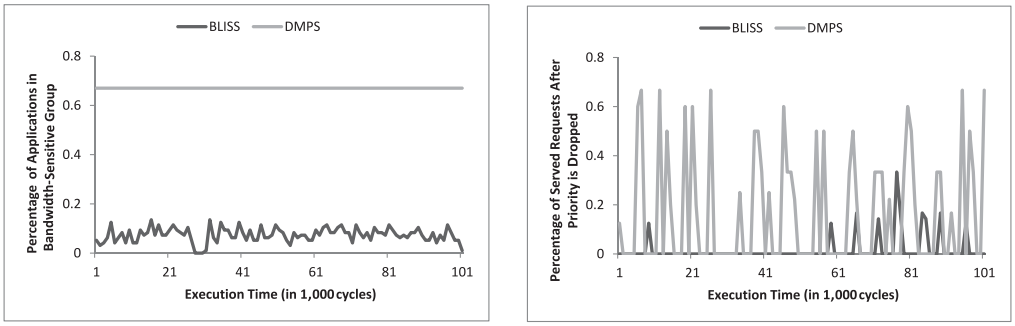
(a) 436.cactusADM (b) 483.xalancbmk

Fig. 9. Request service distribution of DMPS and TCM (applications with higher priority value in TCM have lower priority).

9.0% and 4.3% on the 24-core and 1-channel system. TCM achieves the best system performance in all situations. TCM's system performance benefits over DMPS increase slightly when the system becomes more bandwidth constrained, whereas the fairness of TCM is much worse than that of DMPS. On the 24-core and 1-channel system with the most severe memory interference, DMPS has 5.2% lower system performance but 9.8% better fairness than TCM. DMPS also significantly outperforms FRFCFS, FRFCFS-Cap, PARBS, and BLISS in terms of system performance and provides high fairness at the same time. Hence, DMPS can achieve high system performance and fairness stably at different core counts and channel counts, and can mitigate memory interference effectively even in the systems that are highly bandwidth constrained.

6.4. Comparison with TCM

To compare TCM to DMPS in terms of fairness, we run a workload with 50% memory intensity in the default 24-core and 4-channel system, and sample the number of requests served for each application every 1,000 cycles. Both the epoch for DMPS and the shuffle interval for TCM are set to 1,000 cycles temporarily. Figure 9 shows the request service distribution for cactusADM and xalancbmk and the priority of two benchmarks in TCM during a 100,000-cycle period. In the 100,000-cycle period, cactusADM and xalancbmk are classified as bandwidth-sensitive applications, and their request service distributions can reflect to some degree whether the fairness is good or bad. On one hand, TCM has high variance in the number of requests served over time for both cactusADM and xalancbmk. This is because TCM ranks individual applications in a total order and prioritizes some bandwidth-sensitive applications over other bandwidth-sensitive applications. Many requests are served during a few intervals when the priority value is very low, corresponding to the spikes, whereas during most of the intervals when the priority value is high, only a small number of requests are served, resulting in very slow progress. Sometimes none of the requests are served for several intervals. Thus, most bandwidth-sensitive applications experience high slowdown in TCM. On the other hand, the request service distribution of cactusADM and xalancbmk in DMPS is less spiky and more steady and the number of requests served is nonzero in nearly all intervals. DMPS assigns the same initial priority to bandwidth-sensitive applications and only deprioritizes applications that have too many requests served to alleviate memory interference. Therefore, bandwidth-sensitive applications can experience lower memory stall times and are progressed in a relatively even and fast pace. Therefore, DMPS can achieve better fairness than TCM.



(a) Ability to classify applications (b) 436.cactusADM

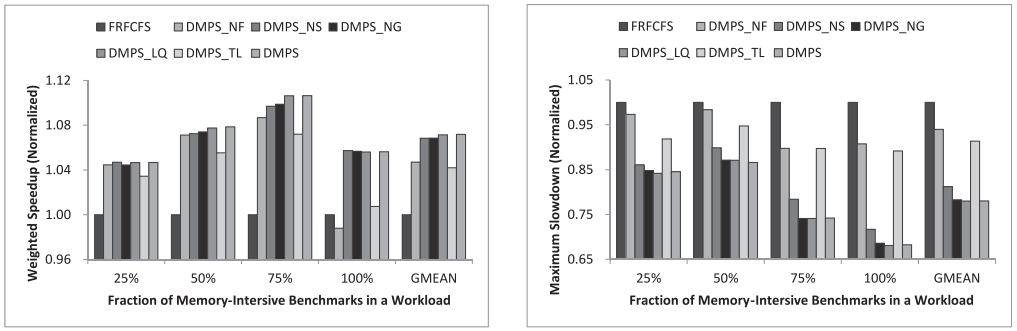
Fig. 10. Comparison with BLISS.

6.5. Comparison with BLISS

We rerun the workload with 50% memory intensity used in Section 6.4 and focus on the difference in applications' classification between BLISS and DMPS. Figure 10(a) shows the percentage of applications in bandwidth-sensitive group during a 100,000-cycle period. The percentage of blacklisted applications in BLISS is less than 20%, and most bandwidth-sensitive applications are mixed with latency-sensitive applications, degrading system performance quite a bit, whereas the percentage of applications in bandwidth-sensitive group is fixed at 67% for DMPS, and latency-sensitive applications are separated from bandwidth-sensitive applications. Thus, DMPS is more effective in the classification of applications. CactusADM is a memory-intensive benchmark in the workload, and Figure 10(b) shows the percentage of requests served for cactusADM after its priority is dropped. During few intervals when cactusADM is blacklisted by BLISS, the percentage of requests served after being blacklisted is less than 40%. During the 100,000-cycle period, cactusADM is classified as bandwidth-sensitive by DMPS. The percentage of requests served after the priority of cactusADM is lowered is more than 40% in many intervals. The apparent discrepancy is because BLISS and DMPS use different criterions to deprioritize an application. BLISS blacklists applications that have a relatively large number of consecutive requests served, whereas DMPS deprioritizes applications that have a relatively large number of total requests served. BLISS cannot control the number of requests served from an application before the application is blacklisted, but DMPS can. Compared to BLISS, DMPS adjusts the priority of interference-causing applications relatively quicker to mitigate memory interference. Hence, DMPS can achieve higher system performance than BLISS.

6.6. Benefits of Different Components

Figure 11 shows the system performance and fairness across 80 workloads for schedulers with one component disabled compared to DMPS. DMPS_NF sets the length of the epoch equal to that of the quantum and disables the fine-grain dynamic priority. In the 100% category, DMPS_NF performs worse than FRFCFS. Compared to DMPS, DMPS_NF degrades system performance and fairness by 2.4% and 20%. DMPS_NS sets the length of the quantum to 100 million cycles (maximum simulation time), so DMPS_NS does not synchronize across memory controllers. DMPS_NS has 0.4% lower system performance and 4% higher unfairness than DMPS. DMPS_NG does not classify applications into two groups at every quantum and has 0.4% lower system performance and 0.3% higher unfairness than DMPS. This is because grouping of applications works only for heterogeneous applications. The system performance benefits



(a) System performance (b) Fairness

Fig. 11. Benefits of different components in DMPS. (DMPS_NF: no fine-grain epoch; DMPS_NS: no synchronization; DMPS_NG: not to classify applications; DMPS_LQ: use characteristics in the last quantum to classify applications; DMPS_TL: only two priority levels).

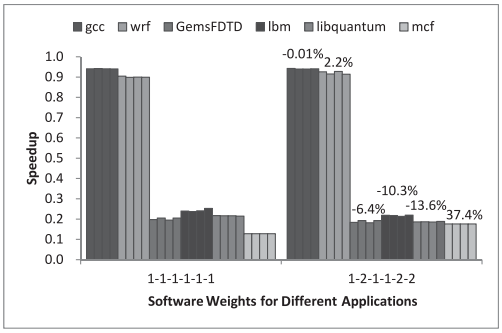
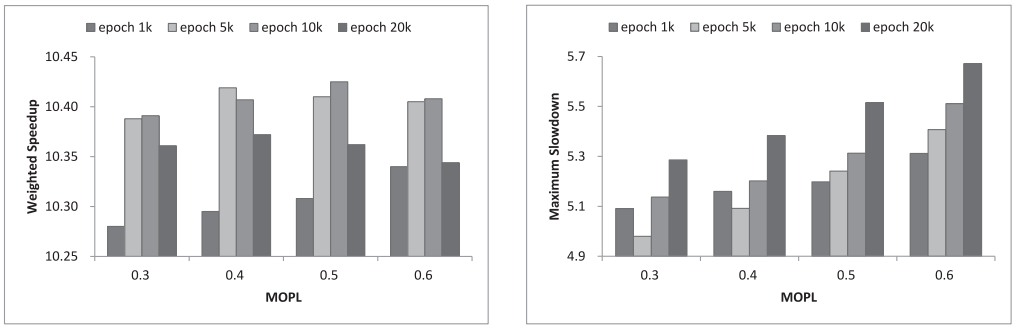


Fig. 12. Performance of applications with different software weights.

from grouping applications are more obvious in the 50% and 75% categories. DMPS_LQ classifies applications using the characteristics in the last quantum. Compared to DMPS, DMPS_LQ has 0.1% lower system performance and 0.1% better fairness than DMPS. The only difference between DMPS_TL and DMPS is the number of priority levels. DMPS_TL has two priority levels, and bandwidth-sensitive applications have the same priority over the epoch. Compared to DMPS, DMPS_TL has 2.8% lower system performance and 17% unfairness. Therefore, we can conclude that DMPS benefits more from the fine-grain epoch to reset priority and the dynamic multilevel priority.

6.7. Sensitivity to Software Weights

Although memory occupancy is a simple metric to measure memory interference, DMPS does not lose the control to applications' progress like BLISS. Figure 12 shows the performance of applications for DMPS with different weights. The software weights of wrf, libquantum, and mcf are improved from 1 to 2, and that of gcc, GemsFDTD, and lbm remain unchanged. We can make three observations. First, gcc and wrf are latency-sensitive applications, and their performance has no significant diversification. Since the software weight of wrf is higher, the performance of wrf improves by 2.2% and the performance of gcc degrades by 0.01%. As we expected, assigning high software weights to bandwidth-sensitive applications has no significant influence on the performance of latency-sensitive applications. Second, software weights influence the performance of bandwidth-sensitive applications quite a bit. The performance of GemsFDTD, lbm,



(a) System performance (b) Unfairness

Fig. 13. Sensitivity to algorithm parameters.

and libquantum degrades by 6.4%, 10.3%, and 13.6%, respectively, and that of mcf improves by 37.4%. Libquantum and mcf are both very memory intensive; libquantum has high row-buffer locality, whereas mcf's row-buffer locality is very low. Since mcf has the same software weight as libquantum, mcf breaks the row-buffer locality of libquantum and delays libquantum quite a bit. Third, the performance benefits are not linear with software weights: having higher software weights does not equate to higher performance benefits. Thus, it remains impossible to accurately control applications' progress now. But this problem is what we plan to solve in the future.

6.8. Sensitivity to Algorithm Parameters

Figure 13 shows the system performance and fairness of DMPS for different values of *MOPL* and the epoch across 80 default workloads. An epoch of 5,000 cycles and an *MOPL* of 0.3 provide a good balance between system performance and fairness. To avoid the complicated division in hardware implementation, DMPS prioritizes applications into multiple levels by comparing the number of read requests served to the per-level threshold *ReqPL* instead of comparing memory occupancy to *MOPL*. As shown in Algorithm 1, the per-level threshold *ReqPL* is related to the product of the epoch and *MOPL*. If the per-level threshold is too large, interference-causing applications have a long time to serve lots of read requests before they are penalized to a lower level, and the dynamic property of priority is weakened to the static property to some degree, resulting in high interference and unfairness. If the per-level threshold is too small, priority levels of all applications are easily lowered, and latency-sensitive applications cannot stay at a high priority level for a long time, degrading system performance significantly. Hence, the per-level threshold is important to balance system performance and fairness.

6.9. Sensitivity to Priority Levels

Figure 14 shows system performance and fairness of DMPS by varying the number of priority levels with different values of *MOPL* and fixed 5,000 cycles of an epoch. First, three priority levels can provide better system performance and fairness. More priority levels will hinder the bandwidth-sensitive applications from sharing memory bandwidth fairly, whereas fewer priority levels are not able to effectively mitigate the interference among bandwidth-sensitive applications. System performance increases significantly from two levels to three levels, and fairness decreases fast from three levels to five levels. Second, the system performance and fairness increase at first and then decrease with the value of *MOPL* for the same priority level. Too small and too large *MOPL* cannot classify applications into different priority levels effectively. Third,

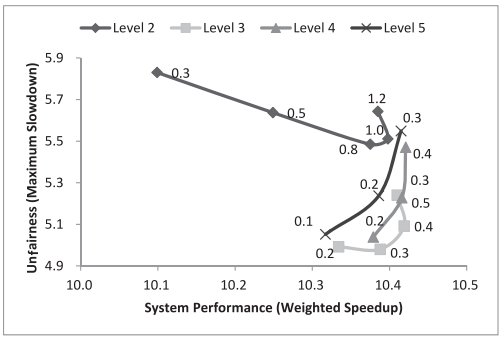
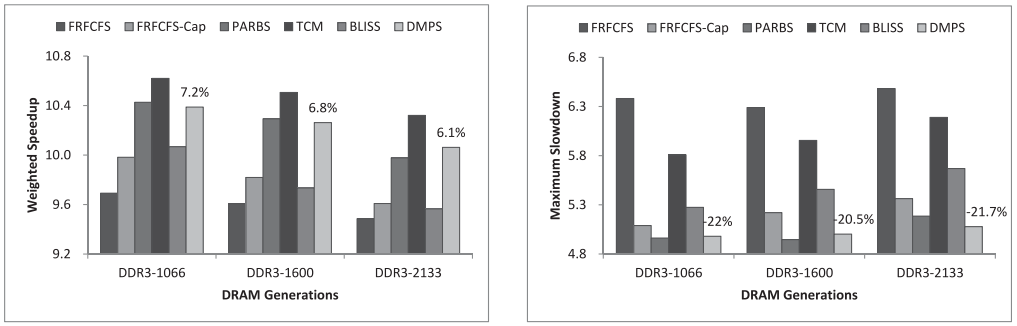


Fig. 14. Sensitivity to different priority levels.



(a) System performance (b) Unfairness

Fig. 15. Sensitivity to memory technology.

the value of *MOPL* for the optimal point becomes lower as the number of priority levels increases: 1.0 for two-level DMPS, 0.3 for three-level DMPS, 0.3 for four-level DMPS, and 0.2 for five-level DMPS. Since having more levels equates to higher hardware complexity and cost, it is clear that three levels with an *MOPL* of 0.3 provide a good balance among system performance, fairness, and hardware complexity.

6.10. Sensitivity to Memory Technology

Figure 15 shows the system performance and fairness across 80 default workloads for different DRAM models. DRAM models DDR3-1600 and DDR3-2133 are added into our simulation infrastructure. First, the system performance and fairness of all schedulers decrease when the frequency of DRAM increases. This is because the value of timing parameters (e.g., t_{RCD}, t_{RP}, t_{CL}) increases and the simulation time is fixed at 100 million cycles. Second, the benefits of DMPS over FRFCFS decrease when the frequency of DRAM increases. Although newer and faster memory technologies can provide high bandwidth and alleviate much of the memory interference, DMPS still has 6.1% higher weighted speedup and 21.7% lower maximum slowdown than FRFCFS at DDR3-2133. Third, compared to other schedulers, DMPS performs better for faster memory technologies. DMPS can outperform PARBS at DDR3-2133 in terms of system performance and fairness. The gap of system performance between TCM and DMPS becomes smaller. Therefore, DMPS has a stable and high system performance and fairness even with newer and faster memory technologies.

7. CONCLUSIONS

We introduce DMPS, a new and simple memory access scheduler based on dynamic multilevel priority. Most previous application-aware schedulers pursue high system performance and fairness and employ a per-application ranking mechanism, incurring high hardware complexity and cost. The latest application-aware scheduler pursues low hardware complexity and cost, and employs a per-group ranking scheme to ensure quick scheduling actions, severely degrading system performance and fairness. DMPS provides a good balance among system performance, fairness, and hardware complexity. By introducing memory occupancy to measure the ability to cause memory interference, DMPS can simply classify applications into two groups. To improve system performance, latency-sensitive applications are prioritized over bandwidth-sensitive applications. To ensure fairness across applications, DMPS dynamically prioritizes applications into multiple levels, and applications with high enough memory occupancy are deprioritized. We evaluated DMPS across 80 workloads on different system configurations and compared it to five previous schedulers. The evaluation result shows that DMPS has a significantly higher system performance and fairness than FRFCFS, FRFCFS-Cap, and the per-group ranking scheduler BLISS. Compared to per-application ranking schedulers, DMPS reduces the hardware complexity greatly and achieves high fairness while slightly degrading system performance. Thus, we conclude that DMPS can achieve high system performance and fairness with low hardware complexity, which is more suitable for current and future multicore systems.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable feedback, which immensely helped in improving the article.

REFERENCES

Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. 2012. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'12)*.

Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth H. Pugsley, Aniruddha N. Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. 2012. USIMM: The Utah simulated memory module. Available at https://www.cs.utah.edu/~rajeev/pubs/usimm.pdf.

Niladrish Chatterjee, Mike O'Connor, Gabriel H. Loh, Nuwan Jayasena, and Rajeev Balasubramonia. 2014. Managing DRAM latency divergence in irregular GPGPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'14)*.

Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. 2013. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'13)*.

Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R. Das. 2009. Application-aware prioritization mechanisms for on-chip networks. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*.

Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R. Das. 2010. Aérgia: Exploiting packet latency slack in on-chip networks. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'10)*.

Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2010. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*.

Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, José A. Joao, Onur Mutlu, and Yale N. Patt. 2011. Parallel application memory scheduling. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*.

Stijn Eyerman and Lieven Eeckhout. 2009. Per-thread cycle accounting in SMT processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*.

Saugata Ghose, Hyodong Lee, and José F. Martínez. 2013. Improving memory scheduling via processor-side load criticality information. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'13)*.

Boris Grot, Stephen W. Keckler, and Onur Mutlu. 2009. Preemptive virtual clock: A flexible, efficient, and cost-effective QOS scheme for networks-on-chip. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*.

John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4, 1–17.

Ibrahim Hur and Calvin Lin. 2004. Adaptive history–based memory schedulers. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-37)*.

Takakazu Ikeda, Shinya Takamaeda-Yamazaki, Naoki Fujieda, Shimpei Sato, and Kenji Kise. 2012. Request density aware fair memory scheduling. In *Proceedings of the IEEE Multiconference on Systems and Control (MSC'12)*.

Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'08)*.

Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. 2012a. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *Proceedings of the Annual Design Automation Conference (DAC'12)*.

Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, Michael Sullivan, Ikhwan Lee, and Mattan Erez. 2012b. Balancing DRAM locality and parallelism in shared memory CMP systems. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'12)*.

Samira Khan, Alaa R. Alameldeen, Chris Wilkerson, Onur Mutlu, and Daniel A. Jiménez. 2014. Improving cache performance by exploiting read-write disparity. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*.

Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. 2014. Bounding memory interference delay in COTS-based multi-core systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*.

Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. 2016. Bounding and reducing memory interference in COTS-based multi-core systems. *Real-Time Systems* 52, 3, 356–395.

Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010a. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'10)*.

Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010b. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*.

Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer Architecture Letters* 15, 1, 45–49.

Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. 2012. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*.

Kun Luo, Jayanth Gummaraju, and Manoj Franklin. 2001. Balancing throughput and fairness in SMT processors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'01)*.

Asit K. Mishra, Onur Mutlu, and Chita R. Das. 2013. A heterogeneous multiple network-on-chip design: An application-aware approach. In *Proceedings of the Annual Design Automation Conference (DAC'12)*.

Thomas Moscibroda and Onur Mutlu. 2007. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of the USENIX Security Symposium (SS'07)*.

Thomas Moscibroda and Onur Mutlu. 2008. Distributed order scheduling and its application to multi-core DRAM controllers. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'08)*.

Janani Mukundan and José F. Martínez. 2012. MORSE: Multi-objective reconfigurable self-optimizing memory scheduler. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'12)*.

Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. 2011. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*.

Onur Mutlu. 2013. *Memory Scaling: A Systems Architecture Perspective*. Available at https://users.ece.cmu.edu/~omutlu/pub/memory-scaling_imw13.pdf.

Onur Mutlu, Justin Meza, and Lavanya Subramanian. 2015. *The Main Memory System: Challenges and Opportunities*. Available at http://repository.cmu.edu/ece/360/.

Onur Mutlu and Thomas Moscibroda. 2007. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40)*.

Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'08)*.

Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. 2004. Pinpointing representative portions of large Intel® Itanium® programs with dynamic instrumentation. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-37)*.

Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39)*.

Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory access scheduling. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'00)*.

Allan Snavely and Dean M. Tullsen. 2000. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*.

Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. 2014. The blacklisting memory scheduler: Achieving high performance and fairness at low cost. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'14)*.

Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. 2016. BLISS: Balancing performance, fairness and complexity in memory access scheduling. *IEEE Transactions on Parallel and Distributed Systems* 27, 10, 3071–3087.

Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. 2015. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*.

Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. 2013. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'13)*.

Hiroyuki Usui, Lavanya Subramanian, Kai Wei Chang, and Onur Mutlu. 2016. DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *ACM Transactions on Architecture and Code Optimization* 12, 4, Article No. 65.

Mingli Xie, Dong Tong, Kan Huang, and Xu Cheng. 2014. Improving system throughput and fairness simultaneously in shared memory CMP systems via dynamic bank partitioning. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*.

Hongzhong Zheng, Jiang Lin, Zhao Zhang, and Zhichun Zhu. 2008. Memory access scheduling schemes for systems with multi-core processors. In *Proceedings of the International Conference on Parallel Processing (ICPP'08)*.

Danfeng Zhu, Rui Wang, Hui Wang, Depei Qian, Zhongzhi Luan, and Tianshu Chu. 2010. A fair thread-aware memory scheduling algorithm for chip multiprocessor. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'10)*.

Received March 2016; revised August 2016; accepted October 2016