# Providing Predictable Performance via a Slowdown Estimation Model

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

Interapplication interference at shared main memory slows down different applications differently. A few slowdown estimation models have been proposed to provide predictable performance by quantifying memory interference, but they have relatively low accuracy. Thus, we propose a more accurate slowdown estimation model called *SEM* at main memory. First, SEM unifies the slowdown estimation model by measuring IPC directly. Second, SEM uses the per-bank structure to monitor memory interference and improves estimation accuracy by considering write interference, row-buffer interference, and data bus interference. The evaluation results show that SEM has significantly lower slowdown estimation error (4.06%) compared to STFM (30.15%) and MISE (10.1%).

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

Additional Key Words and Pharses: Memory access scheduling, slowdown estimation, bandwidth allocation, predictable performance

**ACM Reference format:**
Dongliang Xiong, Kai Huang, Xiaowen Jiang, and Xiaolang Yan. 2017. Providing Predictable Performance via a Slowdown Estimation Model. *ACM Trans. Archit. Code Optim.* 14, 3, Article 25 (August 2017), 26 pages.
https://doi.org/10.1145/3124451

## 1 INTRODUCTION

In most multicore systems, main memory is shared by applications running on different cores. The heavy contention for the available memory bandwidth destroys row-buffer locality and bank-level parallelism of individual applications, and causes additional address/data bus conflicts. The interapplication interference, if not properly managed, may slow down individual applications and degrade overall system performance significantly. Moreover, the interapplication interference makes it very hard to accurately estimate the slowdown of each application online. The slowdown of an

This work is a new article, not an extension of a conference paper. X. Yan is also with State Key Laboratory of ASIC & System, Fudan University, China.
This research was supported by grant 2017ZX01030-102-002 of the National Science and Technology Major Project.
Authors' addresses: D. Xiong, K. Huang (corresponding author), X. Jiang, and X. Yan, Institute of VLSI Design, Zhejiang University, China; emails: {xiongdl, huangk, jiangxw, and yan}@vlsi.zju.edu.cn.

https://doi.org/10.1145/3124451

application highly depends on its own sensitivity to available shared resources and co-running applications. With the knowledge of accurate estimated slowdown, the operating system and smart resources, such as caches and memory controllers, can allocate shared resources better to enforce quality of service (QoS) and fairness in multicore systems. To this end, two different solution directions have been explored by previous work: (1) to alleviate interference, therefore improving overall system performance and fairness, and (2) to precisely quantify and control the impact of interference on application slowdowns, thereby providing soft QoS guarantees and predictable performance (Mutlu 2013; Mutlu et al. 2015).

Smart resources and oblivious resources are two approaches to mitigate interference at different components of the memory system. In the *smart resources* approach, interconnects (Das et al. 2009; Mishra et al. 2013), caches (Khan et al. 2014), and memory controllers (Mutlu and Moscibroda 2007, 2008; Kim et al. 2010a; Kim et al. 2010b; Subramanian et al. 2014; Xiong et al. 2016) are modified to allocate resources in a QoS-aware manner. Most prior work focuses on the design of QoS-aware memory controllers, with the goal of providing high system performance and fairness. These application-aware schedulers address two unfairness problems of FRFCFS (Rixner et al. 2000) in multicore systems: (1) favoring memory-intensive applications over memory-nonintensive applications, and (2) favoring applications with high row-buffer locality over applications with low row-buffer locality. The key idea is to identify interference-causing applications and mitigate interference by prioritizing vulnerable-to-interference applications. In the *oblivious resources* approach, the resources are not modified to be QoS aware, and their allocation is controlled at the cores or the operating system (Kaseridis et al. 2011; Ebrahimi et al. 2010; Das et al. 2013; Liu et al. 2012; Jeong et al. 2012b; Muralidhara et al. 2011; Xie et al. 2014). By partitioning memory channels/banks among applications, request streams of different applications are isolated to different channels/banks, eliminating row-buffer locality interference. The *smart resources* approach is orthogonal to the oblivious resources approach, and they can be combined to enable more effective interference mitigation.

With the goal of providing soft or hard performance guarantees, few prior works have focused on precisely quantifying and controlling the impact of interference on application slowdowns. Once accurate slowdown estimates are available, the shared resources can be allocated to each application in a slowdown-aware manner, thereby meeting the performance requirements for specific applications and improving the performance and efficiency of the entire system. The slowdown of an application is defined as the ratio of IPC when run alone ($IPC_{alone}$) and IPC when running with other applications ($IPC_{shared}$). The slowdown is also equal to the ratio of the execution time when the application is running with other applications ($ET_{shared}$) and the execution time when the application has run alone on the same system ($ET_{alone}$). Equation (1) shows the relationship between two definitions, where $CI$ is the number of committed instructions when the application is running with other applications and $ET_{interference}$ is the number of interference cycles. $IPC_{shared}$ and $ET_{shared}$ of an application can be directly measured, but $IPC_{alone}$ or $ET_{interference}$ of the application is unknown in the share run. Some prior work (Mars et al. 2011; Eklov et al. 2012) chooses to profile applications offline to estimate their alone performance. However, the offline profile is not feasible or accurate for applications that are heavily input set dependent, and it is hard to get the profile in some environments (Subramanian et al. 2015). Therefore, the key challenge is how to estimate $IPC_{alone}$ or $ET_{interference}$ of each application online in the presence of interapplication interference.

$$Slowdown = \frac{IPC_{alone}}{IPC_{shared}} = \frac{CI/ET_{alone}}{CI/ET_{shared}} = \frac{ET_{shared}}{ET_{alone}} = \frac{ET_{shared}}{ET_{shared} - ET_{interference}} \quad (1)$$

The online mechanisms to estimate slowdown for individual applications are divided in two directions: (1) estimating $ET_{interference}$ (Mutlu and Moscibroda 2007; Eyerman and Eeckhout 2009; Ebrahimi et al. 2010) and (2) estimating $IPC_{alone}$ (Subramanian et al. 2013, 2015). The stall-time fair memory scheduler (STFM) (Mutlu and Moscibroda 2007), per-thread cycle accounting (PTCA) (Eyerman and Eeckhout 2009), and fairness via source throttling (FST) (Ebrahimi et al. 2010) estimate interference at an individual request granularity. They determine the number of cycles by which each request of an application is delayed due to interference. Since memory requests can be served in parallel at shared caches and main memory, it is difficult and inaccurate to quantify interference at a per-request granularity. Instead, memory-interference–induced slowdown estimation (MISE) (Subramanian et al. 2013) and the application slowdown model (ASM) (Subramanian et al. 2015) quantify the interference in an aggregate manner for a large set of requests. The performance is represented by the request service rate in MISE and the cache access rate in ASM. In addition, the main memory interference for an application is minimized by giving the application the highest priority to access memory. Therefore, MISE and ASM have much higher slowdown estimation accuracy. STFM and MISE only consider the interference at main memory, whereas FST, PTCA, and ASM also take into account the interference at shared caches. STFM, MISE, and ASM have one common drawback: the slowdown estimation model for compute-intensive applications is inaccurate, since the request service rate or cache access rate cannot precisely represent these applications' performance. Another drawback of MISE is that the mechanism to identify interference cycles is inaccurate, and MISE does not consider write interference and row-buffer interference.

The goal of this work is to design an online mechanism to accurately estimate the slowdown of each application due to memory interference. To this end, we propose a slowdown estimation model called *SEM*. First, SEM unifies the slowdown estimation model by measuring the performance IPC directly. The request service rate and cache access rate are not proportional to the performance IPC for applications that spend a large fraction of execution time in the compute phase. Moreover, IPC is a more fine-grain metric. We observe that estimating IPC directly can significantly improve the accuracy for compute-intensive applications.

Second, to accurately estimate the alone performance of an application, SEM uses the per-bank structure to monitor interference and considers write interference, row-buffer interference, and data bus interference. SEM minimizes the interference for the estimated application by giving it the highest priority to access memory, similar to MISE. But the interference is not eliminated. SEM divides memory interference into intrabank interference and interbank interference. The intrabank interference includes the write and row-buffer interference in each bank, and the interbank interference is the data bus interference across banks.

Third, SEM considers bank-level parallelism when quantifying the interference cycles. Due to abundant memory-level parallelism, the service of difference requests will likely overlap. The interference in a cycle is estimated as the ratio of the number of banks in which the highest priority requests are interfered and the number of banks in which the highest priority requests are waiting.

This article makes the following contributions:

— We propose SEM, an online model that accurately estimates application slowdowns due to main memory interference. SEM considers write interference, row-buffer interference, and data bus interference.
— We compare IPC to the request service rate and cache access rate, showing that IPC is better than the request service rate and cache access rate in slowdown estimation.
— We compare SEM to STFM and MISE across a wide range of workloads and system configurations. The results show that SEM has much higher accuracy than STFM and MISE in slowdown estimation.

- We compare SEM-QoS to MISE-QoS to provide soft performance guarantees for a single application, showing that SEM-QoS is more robust than MISE-QoS.
- We compare ASMIPC-SEM to ASM-MISE at the shared cache, showing that ASMIPC-SEM has higher estimation accuracy than ASM-MISE in the presence of shared cache and memory interference.

## 2 BACKGROUND

### 2.1 DRAM Basis

In multicore systems, the DRAM-based memory system is organized hierarchically as channels, ranks, and banks. Each channel consists of ranks, and each rank consists of multiple banks operated in lockstep. Channels are independent, but the ranks or banks in a channel share the address, command, and data buses of the channel. Thus, the parallelism among channels is higher than that among ranks and banks. The bank is a two-dimensional data array, organized as rows and columns. Each bank has an internal structure called a *row buffer*, which acts as an interface between the data array and memory controller. The access latency of a memory request is highly dependent on the status of the row buffer: (1) $t_{CAS}$ for a row hit, when the memory request accesses the row that is currently open in the row buffer; (2) $t_{RAS} + t_{CAS}$ for a row closed, when the row buffer has no open row; and (3) $t_{RP} + t_{RAS} + t_{CAS}$ for a row conflict, when the row address of the memory request is different from the open row in the row buffer. The sequent commands to perform a row conflict are PRE, ACT, and read/write. Therefore, start-of-the-art memory schedulers prioritize row-hit requests over row-closed and row-conflict requests to maximize DRAM throughput.

### 2.2 Memory Scheduling

FRFCFS (Rixner et al. 2000) is a commonly used memory scheduling algorithm. To maximize DRAM throughput, FRFCFS leverages the row buffer by prioritizing row hits over row conflicts. When the row hit status is equal, older requests are prioritized over younger requests. However, FRFCFS unfairly favors requests of applications with high row-buffer locality or memory intensity in multicore systems. The interapplication interference can severely degrade overall system performance and fairness. Application-aware memory scheduling is a prevalent solution to mitigate interapplication interference, with the goal of improving system performance and fairness. Typically, application-aware memory schedulers monitor applications' memory access characteristics and rank applications individually so that requests from vulnerable-to-interference applications have higher priority. PARBS (Mutlu and Moscibroda 2008) provides fairness by request batching and preserves intrathread bank-level parallelism by thread ranking. ATLAS (Kim et al. 2010a) ranks applications periodically by attained service to provide scalability and high performance. TCM (Kim et al. 2010b) separates applications into two clusters, ranks applications in latency-sensitive cluster by memory intensity to improve system performance, and periodically shuffles the priority of applications in a bandwidth-sensitive cluster to provide fairness. However, PARBS, ATLAS, and TCM incur high hardware complexity due to ranking applications in a total order. To reduce hardware complexity, BLISS (Subramanian et al. 2014, 2016) dynamically separates applications into two groups and DMPS (Xiong et al. 2016) dynamically prioritizes applications into multiple levels. Some application-unaware schedulers rank requests directly to mitigate memory interference. Ghose et al. (2013) prioritize critical memory requests that potentially stall the instruction window for a long time. AHB (Hur and Lin 2004) matches a command pattern to avoid bottlenecks in the reorder queue and minimizes the expected latency of scheduled operations. RLMS (Ipek et al. 2008) and MORSE (Mukundan and Martínez 2012) employ the Q-learning algorithm in reinforcement learning to optimize long-term performance.

## 2.3 Slowdown Estimation

We only show the prior work that estimates application slowdowns online. STFM (Mutlu and Moscibroda 2007) aims to equalize DRAM-related slowdown experienced by each thread. DRAM-related slowdown is the ratio of the memory stall time when running with other applications and the memory stall time when run alone. STFM has two drawbacks. First, STFM does not take into account the compute phase, which is the major part for memory-nonintensive applications. Second, STFM estimates the slowdown of an application at a per-request granularity when it is receiving significant memory interference. The abundant parallelism in the memory system makes it hard to determine the accurate cycles by which each request is delayed due to memory interference. MISE (Subramanian et al. 2013) uses the request service rate as a proxy for performance, at the assumption of relatively stable phase behavior over a million cycles. To estimate an application's alone request service rate (ARSR), MISE gives the application the highest priority to minimize the interference from other applications. MISE has two drawbacks. First, the equation to estimate the slowdown of memory-nonintensive applications is not right since the performance of memory-nonintensive applications is not proportional to the request service rate. The memory-nonintensive applications spend a large fraction of execution time in the compute phase when the core is not stalled on memory accesses. Second, the mechanism to determine the number of interference cycles is not accurate. MISE does not consider the write interference, row-buffer interference, and bank-level parallelism. Moreover, the simple mechanism overestimates the number of interference cycles during read drain mode.

STFM and MISE only quantify the interference at main memory, whereas PTCA (Eyerman and Eeckhout 2009), FST (Ebrahimi et al. 2010), and ASM (Subramanian et al. 2015) also consider the interference at shared caches. Both PTCA and FST estimate the alone execution time at a per-request granularity. PTCA quantifies the reduction in per-thread memory-level parallelism due to multithreading and estimates the number of additional conflict misses at the shared branch predictor, caches, and TLBs by using auxiliary tag directories. A miss at the shared branch predictor, caches. and TLBs is identified as a conflict miss when it is a hit in the corresponding tag directory. FST uses a mechanism similar to STFM to quantify interference at the main memory. To identify contention misses at shared caches, FST uses a pollution filter for each application to track the cache lines evicted by other applications. An access that misses in the cache and hits in the pollution filer is classified as a contention miss. The goal of ASM is to estimate the alone cache access rate in an aggregate way. ASM applies MISE to estimate interference at main memory. To quantify interference at shared caches, ASM also maintains an auxiliary tag store for each application to track contention misses, but it uses set sampling to significantly reduce the overhead of the auxiliary tag store with negligible loss in accuracy. In this work, we only consider the interference at main memory and leave the interference at shared caches as part of our future work. NAS (Xiang et al. 2016) considers the distributed NoC-level interference and accurately estimates the impact of network delays on application slowdown. FAST (Xiang et al. 2016) uses the slowdown information provided by NAS to throttle NoC nodes, therefore improving system performance and fairness. NAS can be combined with other slowdown models (e.g., MISE, ASM, SEM) to consider interference in the NoC, caches, and main memory holistically.

In COTS-based multicore systems, Kim et al. (2014) bounds the worst-case memory interference delay experienced by a task in a request-driven approach and job-driven approach. Further, Kim et al. (2016) reduces memory interference by co-locating memory-intensive tasks on the same core with dedicated DRAM banks, thereby providing high task schedulability. However, Kim et al. (2016) optimistically assumes that each core is in order with one outstanding memory request, so any delays from shared resources are additive to the task execution times. Kim et al. (2016) also pessimistically assumes that each command of a request can be delayed by all commands that have arrived earlier at other banks. In heterogeneous systems, Jeong et al. (2012a) propose a dynamic QoS policy to meet a GPU target deadline while achieving high CPU performance. The priority of a GPU is equal to or lower than that of a CPU most of the time, and a GPU is prioritized over a CPU only when the GPU is close to a deadline. Thus, the GPU may potentially miss deadlines. DASH (Usui et al. 2016) extends the GPU to hardware accelerators (HWAs) and proposes the distributed priority scheme to tackle the problem of HWAs missing deadlines. DASH prioritizes a HWA any time it is not on track to meet its deadline. However, SEM quantifies the major interference only when requests from the highest priority application are actually blocked by requests from other applications and dynamically allocates bandwidth based on the estimated slowdown to provide soft performance guarantees.

### 2.4 Prefetch Handling Policy

Based on prefetch accuracy, PADC (Lee et al. 2008) prioritizes useful prefetch and demand requests over useless prefetch requests to maximize DRAM throughput, and proactively drops likely useless prefetch requests to reduce the negative effect of useless prefetch requests. Ebrahimi et al. (2011) prioritize demand requests of applications that are not prefetch friendly and memory nonintensive over accurate prefetches, and enable shared resource management techniques to effectively handle prefetches. Further, Nachiappan et al. (2012) distinguish prefetches from different applications in the NoC routers to mitigate the negative effects of prefetching. Lee et al. (2014) combine a prefetch-aware network and congestion-sensitive prefetch control mechanism to improve system performance. Liu et al. (2016) propose an attacking prefetch filter to avoid prefetching-caused interthread invalidations, and develop a thread-aware data prefetching mechanism to tune prefetching aggressiveness, reducing shared resource contention. These prefetch handling policies can be combined with SEM to enable more effective slowdown estimation in the presence of prefetching.

### 2.5 Write Queue Management and Write Scheduling

To reduce write-caused interference, Lee et al. (2010) employ the *drain_when_full* write buffer policy and aggressively send row-hit writebacks before they would normally be evicted by the cache replacement policy. Stuecheli et al. (2010) coordinate last-level cache policy and memory scheduling by the virtual write queue. The virtual write queue increases row-hit writebacks, enables longer write bursts, and improves read/write priority determination. Seshadri et al. (2014) remove the dirty bits from the tag store and organize them by the dirty-block index (DBI). Each entry of the DBI tracks the dirty bit information of all blocks in some DRAM row. The DBI is simple and effective for supporting optimizations such as aggressive DRAM-aware writebacks and bypassing cache lookups. Chang et al. (2014) proactively schedule a per-bank refresh when writes are drained in a batch and parallelize refreshes and accesses within a bank by exploiting the subarray structure. However, commodity DDR DRAM does not support per-bank refresh, and DRAM organizations need to be modified.

## 3 SLOWDOWN ESTIMATION MODEL

### 3.1 Unify Slowdown Estimation via IPC

Generally, an application alternates between the cache/memory phase and compute phase during its life cycle. In the cache phase, the application is stalled on shared cache accesses. In the memory phase, the application is stalled waiting for memory. In the compute phase, the application is not stalled for shared cache or memory accesses. MISE has observed that the performance of a memory-bound application is roughly proportional to the rate at which its memory requests are served. Similarly, ASM has observed that the performance of each application is proportional to the rate at which it accesses the shared cache. However, the cache access rate is only suitable for applications that are sensitive to cache capacity and/or memory bandwidth. For an application that spends a significant fraction of its execution time in the compute phase, the performance will not increase proportionally with the request service rate or cache access rate.

In SEM, we estimate the performance IPC directly. Since IPC contains all information in the cache/memory phase and compute phase, there is no need to identify whether an application is compute intensive or not. The slowdown for compute-intensive and compute-nonintensive applications can be estimated by Equation (2). In other words, the slowdown estimation model is unified by IPC. Moreover, IPC is a more fine-grain metric than the request service rate and cache access rate because the number of committed instructions is much larger than the number of cache/memory accesses served in a time slice. As shown later in Figures 2 and 13, IPC can improve the estimation accuracy for compute-intensive applications, regardless if the shared cache is enabled because the request service rate and cache access rate are just proxies for the performance IPC.

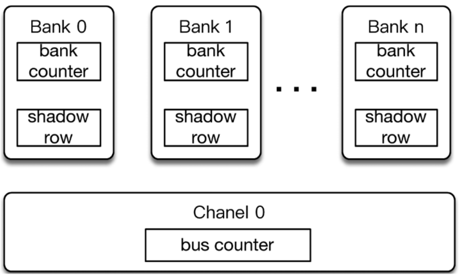
$$Slowdown = \frac{IPC_{alone}}{IPC_{shared}} \tag{2}$$

To effectively detect phase changes in shared cache and main memory behavior, SEM divides execution time into multiple quanta of length $T_{quantum}$ cycles (a few million processor cycles). At the end of each quantum, SEM measures $IPC_{shared}$, estimates $IPC_{alone}$, and computes the slowdown of each application as the ratio of the application's $IPC_{alone}$ and $IPC_{shared}$. $IPC_{shared}$ of an application is the rate at which the application's instructions are committed while it is running with other applications. Each processor can simply keep a counter that tracks the number of committed instructions for the application during each quantum and transfer the counter to the metacontroller at the end of each quantum. The $IPC_{shared}$ for each application can be computed as Equation (3), where $CI_{shared}$ is the number of committed instructions for the application during a quantum.

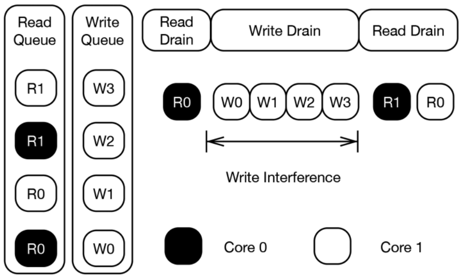
$$IPC_{shared} = \frac{CI_{shared}}{T_{quantum}} \tag{3}$$

However, it is a challenge to estimate $IPC_{alone}$ of an application while it is running with other applications. Preventing other applications from accessing the shared cache and main memory can provide an accurate estimate of $IPC_{alone}$, but other applications will be significantly slowed down, leading to very poor system performance. Instead, we choose to measure an approximate value $IPC_{hp}$, the performance when the application is given the highest priority in accessing memory. When there are no requests from the highest-priority application in the request buffer, the scheduled requests from other applications may change the state of DRAM banks and delay later requests from highest-priority application due to timing constraints. Thus, the memory interference received by an application is only minimized by giving it the highest priority but not eliminated. $IPC_{alone}$ of an application is computed as Equation (4) at the end of each quantum. The corresponding processor keeps track of the number of committed instructions $CI_{hp}$ when the application is given the highest priority and sends the value to the metacontroller at the end of each quantum. $T_{int}$ is the number of interference cycles during the application's highest-priority cycles $T_{hp}$.

$$IPC_{alone} \approx IPC_{hp} = \frac{CI_{hp}}{T_{hp} - T_{int}} \tag{4}$$



(a) Per-bank structure of SEM



(b) Write Interference

Fig. 1. SEM architecture and write interference.

To estimate the alone performance for all co-running applications, SEM further divides each quantum into small epochs of length $T_{epoch}$ cycles (thousands of processor cycles) and assigns an application the highest priority during each epoch. The epochs can be assigned to each application in a round-robin manner. However, SEM employs the lottery scheduling algorithm in MISE to enforce a bandwidth allocation. During each quantum, SEM assigns multiple epochs to each application and these epochs are almost evenly distributed. Prior work (Sherwood et al. 2002; Isci and Martonosi 2003) has observed that applications' phase behavior is relatively stable in a few million processor cycles, so the average highest-priority performance of an application in its assigned epochs can provide a good estimate of its alone performance in a quantum.

## 3.2 Identification of Memory Interference

At the beginning of each epoch, SEM assigns an application the highest priority to access main memory. Measuring $CI_{hp}$ and $T_{hp}$ for the application is straightforward. The most important thing is how to quantify the interference received by the application in the memory controller. Based on DRAM's organization, SEM divides the interference into interbank interference and intrabank interference. Therefore, SEM consists of multiple bank interference monitors and a channel interference monitor in each memory controller, as shown in Figure 1(a).

The interbank interference comes from the contention for the address, command, and data bus shared by banks. When a read/write command is scheduled, it keeps the data bus busy for $t_{Burst}$ DRAM cycles. During these cycles, no other read/write commands can be scheduled even though they might be ready. Since the row activation command is a relatively energy-intensive operation, the timing parameters $t_{RRD}$ and $t_{FAW}$ are defined to constrain the activity rate of DRAM devices, therefore limiting the power consumption of DRAM devices. SEM only considers the interference on the data bus now. The channel interference monitor keeps a counter $bustime$ that tracks the left cycles during which the previous read/write command controls the data bus. When a read/write command for requests from other applications is scheduled, the counter will be set to $t_{Burst}$. The counter decreases by one every DRAM cycle if it is over zero. Thus, the interference on the data bus happens when the counter is over zero and there are waiting read/write commands from the highest-priority application in the request buffer.

The intrabank interference comes from the contention for the shared row-buffer and the broken row-buffer locality. The shared row buffer is the interface between data array and memory controller. Since the data can only be read from or written to the row buffer, requests in the same bank are progressed in serial. When an activate command is scheduled, it keeps the row buffer open for at least $t_{RAS}$ DRAM cycles. During these cycles, the memory controller only

**ALGORITHM 1:** Update of the Interbank and Intrabank Counter

```
Input: hpcid (id of the highest-priority application)
Output: bustime (timing constraints caused by interfering commands on the data bus) and
        banktime[$N_{bank}$] (timing constraints caused by interfering commands in each bank)
repeat
    $bustime = bustime > 0\ ?\ bustime - 1 : 0$;
    for $i = 0$ to $N_{bank} - 1$ do
        $banktime[i] = banktime[i] > 0\ ?\ banktime[i] - 1 : 0$;
    end
    if A command cmd of application id in bank b is scheduled then
        if $id\ ! = hpcid$ then
            if cmd == activate command then
                $banktime[b] = t_{RAS}$;
            end
            else if cmd == precharge command then
                $banktime[b] = t_{RP}$;
            end
            else if cmd == read command then
                $banktime[b] = max(t_{RTP}, banktime[b])$;
                $bustime = t_{Burst}$;
            end
            else if cmd == write command then
                $banktime[b] = max(t_{CWL} + t_{Burst} + t_{WR}, banktime[b])$;
                $bustime = t_{Burst}$;
            end
        end
        else if request is an additional conflict then
            if cmd == activate command then
                $banktime[b] = t_{RCD}$;
            end
            else if cmd == precharge command then
                $banktime[b] = t_{RP}$;
            end
        end
    end
until End of quantum;
```

schedules the ready read/write commands to the open row in the row buffer and blocks the commands to the rows different from the row in the row buffer. As a result, when a request from the highest-priority application arrives in a bank, the request might wait some cycles for the satisfaction of timing constraints before its command is issued. Even if the command for a request from the highest-priority application is scheduled, the latency of the request may be longer than that if the application had been run alone. A request, which results in a row hit with latency $t_{CL}$ if the application was running alone, may result in a row conflict with latency $t_{RP} + t_{RCD} + t_{CL}$ or row closed with latency $t_{RCD} + t_{CL}$ in a shared system. However, the interference between two applications may be positive. If the applications share data, a request may be a row hit in the shared system, whereas it had been a row conflict when the application was running alone. This positive interference is not considered since the applications do not share data in our system. When a request was a row conflict had the application run alone, it may

**ALGORITHM 2:** Quantification of Interference Cycles

```
Input: hpcid (id of the highest-priority application), $drain\_writes$(1: write drain, 0: read drain), $bustime$
       (timing constraints caused by interfering commands on the data bus) and $banktime[N_{bank}]$ (timing
       constraints caused by interfering commands in each bank)
Output: $T_{int}[N_{core}]$ (number of interference cycles of each application in a quantum)
$T_{int}[N_{core}] = 0$;
repeat
    if No commands from application hpcid are scheduled then
        $blpcnt = 0$; // Number of banks in which requests from application hpcid are waiting
        $intcnt = 0$; // Number of banks in which requests from application hpcid are interfered
        if $drain\_writes == 1$ then
            for $i = 0$ to $N_{bank} - 1$ do
                if there are waiting write requests from application hpcid in bank i then
                    $blpcnt++$;
                    if $banktime[i] > 0$ or ($bustime > 0$ and waiting command is write) then
                        $intcnt++$;
                    end
                end
            end
            if there are waiting read requests from application hpcid in request buffer then
                $T_{int}[hpcid] = blpcnt > 0\ ?\ T_{int}[hpcid] + intcnt/blpcnt : T_{int}[hpcid] + 1$;
            end
        end
        else
            for $i = 0$ to $N_{bank} - 1$ do
                if there are waiting read requests from application hpcid in bank i then
                    $blpcnt++$;
                    if $banktime[i] > 0$ or ($bustime > 0$ and waiting command is read) then
                        $intcnt++$;
                    end
                end
            end
            $T_{int}[hpcid] = blpcnt > 0\ ?\ T_{int}[hpcid] + intcnt/blpcnt : T_{int}[hpcid]$;
        end
    end
until End of quantum;
```

result in a row closed in the shared system, and the latency is reduced by $t_{RP}$. At this moment, SEM is not able to account for this positive interference. To identify the negative interference mentioned previously, the bank interference monitor keeps a counter *banktime* that tracks the timing constraints caused by requests from other applications and additional conflicts of the highest-priority application, such as STFM and FST. A *shadow_row* register is maintained for each thread for each bank, which contains the address of the last accessed row by that thread in that bank. As shown in Algorithm 1, when the command for a request from other applications is scheduled, the intrabank counter will be set to $t_{RP}$ for a precharge command, $t_{RAS}$ for an activate command, maximum of $t_{RTP}$ and the counter itself for a read command, and maximum of $t_{CWL} + t_{Burst} + t_{WR}$ and the counter itself for a write command. When the command for a request from the highest-priority application is scheduled, the per-bank counter will be set to $t_{RP}$ for a precharge command and $t_{RCD}$ for an activate command. The intrabank interference

Table 1. Additional Hardware Cost per Memory Controller for SEM

| Storage | Function | Size (bits) |
|---|---|---|
| *hpcid* | Id of the highest-priority application | 2 |
| $T_{hp}$ | Number of highest-priority cycles in a quantum | $20N_{core}$ |
| $T_{int}$ | Number of interference cycles during highest-priority epochs | $20N_{core}$ |
| $T_{quantum}$ | Length of quantum | 20 |
| *bustime* | Interference cycles on data bus | 4 |
| *banktime* | Interference cycles in each bank | $8N_{bank}$ |
| Shadow_row | Address of last accessed row by an application | $16N_{core}N_{bank}$ |
| **Logic** | **Function** | **Number** |
| 4-bit adder | Decrease bustime counter | 4 |
| 8-bit adder | Decrease banktime counter | $8N_{bank}$ |
| 20-bit adder | Aggregate number of interference and highest-priority cycles | $2N_{core}$ |
| 16-bit comparator | Update row-buffer interference | $N_{core}N_{bank}$ |

happens when the per-bank counter is over zero and there are waiting read/write requests from the highest-priority application.

## 3.3 Quantification of Memory Interference

Figure 1(b) shows an example of write interference, where application 0 is given the highest priority. The write drain mode is enabled after R0 from application 0 is served. During write drain cycles, writes from application 1 are served sequently, as there are no writes from application 0 in the write queue. R1 from application 0 can be served until the write drain is done, but it could be served after R0 from application 0 when application 0 has run alone. Thus, the write drain cycles are interference cycles for application 0.

Algorithm 2 shows the quantification of interference cycles of each application. SEM only estimates the interference when there are read requests from the highest-priority application in the read queue. Due to abundant parallelism in the memory system, the service of requests will likely overlap. Thus, SEM considers bank-level parallelism when quantifying the interference in each cycle. When there are highest-priority write requests in write drain mode or there are highest-priority read requests in read drain mode, the interference in a cycle is computed as the ratio of *blpcnt* and *intcnt*. *blpcnt* is the number of banks in which there are pending requests from the highest-priority application, and *intcnt* is the number of banks in which requests from the highest-priority application are blocked. When there are no highest-priority write requests in write drain mode, the interference in a cycle is quantified as 1. The highest-priority application is not interfered with when it has no outstanding read requests.

## 4 IMPLEMENTATION AND HARDWARE COST

On a four-core and one-channel system, Table 1 shows the additional storage and logic of SEM over FRFCFS in each memory controller. First, to estimate alone performance, the memory controller needs a 2-bit register to store *hpcid*, a 20-bit counter per application to store the number of highest-priority cycles, and a 20-bit counter per application to store the number of interference cycles for each application during its highest-priority epochs. Second, to measure shared performance, the memory controller only needs a 20-bit register to store the length of quantum since each application has the same running time. Third, to identify the interference cycles, the memory controller needs a 4-bit counter to track the interference on the data bus, an 8-bit counter per

Table 2. Configuration for the Baseline System

| | |
|---|---|
| Processor | 4–16 cores, 2.13GHz, 4-wide fetch/retire, pipeline depth 10, 160-entry instruction window, private cache 512KB |
| Last-level cache | 4MB, shared, 16-way associative, LRU, block size = 64B, latency = 20 cycles |
| Memory controller | 128-entry read/write queue, write drain watermark 80/40, FRFCFS, open page policy, row-interleaving address mapping |
| Main memory | DDR3-1066 (8-8-8), 4Gbx4 devices, 1 channel, 1 rank/channel, 8 banks/rank, 16KB row buffer |

Table 3. Timing Parameters for a 4Gbx4 1,066Mbps DDR3 DRAM Device

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

bank to track the interference inside each bank, and a 16-bit register per bank per application to maintain the last accessed row. The *bustime* and *banktime* counters decrease by one every cycle if their value is over zero. Finally, to quantify the interference cycles, the memory controller needs a divider to calculate the amount of interference and an adder to aggregate the number of interference cycles. It may be a little complex in hardware to perform bank-level parallelism every cycle. To reduce hardware complexity, you can also divide the number of interference cycles by the average bank-level parallelism at the end of quantum.

## 5 METHODOLOGY AND METRIC

### 5.1 System Configuration

We evaluate SEM using the memory system simulator USIMM (Chatterjee et al. 2012) for the Memory Scheduling Championship (MSC). The USIMM simulator can model a DRAM-based memory system in detail with all required timing parameters. The baseline configuration of the processors and memory system is shown in Table 2, and DRAM main memory is the only shared resource to isolate the memory interference (the last-level cache is not enabled on the baseline configuration). The DRAM device used is DDR3 4Gbx4 running at 1,066Mbps, and Table 3 shows its timing parameters. The memory controller has a centralized read queue and write queue to store read and write requests, respectively. We slightly modify the write drain policy inside USIMM. Read drain mode is enabled by default, whereas write drain mode is enabled only when the length of the write queue is over the high watermark. The scheduler does not exit write drain mode until the length of the write queue is below the low watermark. As for refresh policy, eight refreshes are forced to issue at the end of every $8 \times t_{REFI}$ window. We employ an open page policy to manage the row buffer, 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 open row. The address mapping policy is row interleaving. To compare to ASM, we also add a 4MB shared cache before the main memory. The shared cache is 16-way associative and employs an LRU policy. The access latency to the shared cache is 20 cycles.

Table 4. Memory and Cache Access Behavior of Benchmarks

| Benchmark | MPKI | RBHR | CHR | Benchmark | MPKI | RBHR | CHR |
|---|---|---|---|---|---|---|---|
| 444.*namd* | 0.120 | 0.872 | 0.635 | 482.*sphinx3* | 13.945 | 0.821 | 0.232 |
| 447.*dealII* | 0.156 | 0.812 | 0.689 | 437.*leslie3d* | 17.121 | 0.771 | 0.314 |
| 403.*gcc* | 0.224 | 0.612 | 0.547 | 483.*xalancbmk* | 18.561 | 0.770 | 0.903 |
| 458.*sjeng* | 0.606 | 0.046 | 0.558 | 471.*omnetpp* | 19.693 | 0.784 | 0.979 |
| 445.*gobmk* | 0.880 | 0.475 | 0.740 | 433.*milc* | 20.236 | 0.840 | 0.261 |
| 435.*gromacs* | 1.120 | 0.792 | 0.670 | 459.*GemsFDTD* | 31.425 | 0.592 | 0.430 |
| 464.*h264ref* | 1.866 | 0.884 | 0.964 | 462.*libquantum* | 33.387 | 0.988 | 0.182 |
| 456.*hmmer* | 5.487 | 0.641 | 0.985 | 450.*soplex* | 41.362 | 0.826 | 0.746 |
| 401.*bzip2* | 5.802 | 0.771 | 0.975 | 470.*lbm* | 46.641 | 0.747 | 0.428 |
| 436.*cactusADM* | 6.206 | 0.244 | 0.358 | 434.*zeusmp* | 50.570 | 0.716 | 0.559 |
| 473.*astar* | 8.881 | 0.609 | 0.419 | 429.*mcf* | 88.884 | 0.230 | 0.333 |

MPKI: Misses per kilo instructions (including write requests).
RBHR: Row-buffer hit rate; CHR: cache access hit rate.

## 5.2 Workload Configuration

The benchmarks to drive the USIMM simulator are the memory traces of SPEC CPU2006 (Henning 2006) provided in Ramulator (Kim et al. 2015). Table 4 shows the misses per kilo-instruction (MPKI) and row-buffer hit rate (RBHR) of each benchmark when it is run alone on the baseline system. The cache access hit rate (CHR) for each benchmark is also shown when the shared cache is enabled. Based on these memory access characteristics, we classify the benchmarks into four categories. The thresholds for MPKI and RBHR are 10.0 and 0.75, respectively. The memory intensity of a workload is defined as the percentage of memory-intensive benchmarks in the workload. We form 80 four-core multiprogrammed workloads by varying memory intensity from 0%, 25%, 50%, 75%, to 100% and selecting benchmarks randomly. When the shared cache is enabled, we form workloads by considering MPKI and the CHR. Each workload is run for 100 million representative cycles.

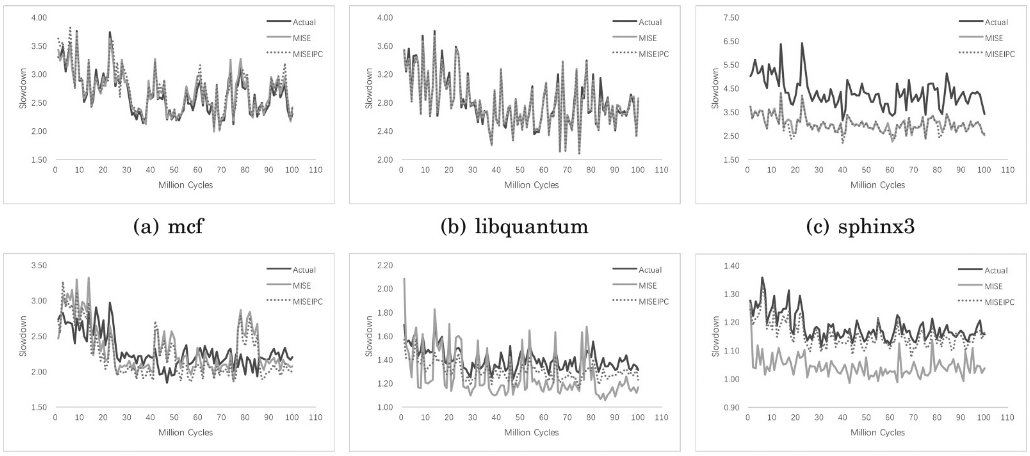
## 5.3 Evaluation Metrics

We use the average estimation error to compare the accuracy of different slowdown estimation models. At the end of each quantum, the estimation error for each application is computed as Equation (5), like Subramanian et al. [2015].When the maximum simulation time is expired, we compute the average estimation error across all quanta as the final estimation error for the applications in a workload. $IPC_{alone}$ is computed as the ratio of the work completed in the shared run for each quantum and the execution time to finish the same amount of work in the alone run. We compute the actual slowdown as the ratio of $IPC_{alone}$ and $IPC_{shared}$.

$$EstimationError = \frac{|IPC_{hp} - IPC_{alone}|}{IPC_{alone}}, ActualSlowdown = \frac{IPC_{alone}}{IPC_{shared}} \tag{5}$$

## 5.4 Parameters for Evaluated Schedulers

FRFCFS has no parameters. To compare the accuracy of different slowdown estimation models, the length of quantum is set to 1 million cycles and the length of epoch is set to 10,000 cycles for STFM, MISE, ASM, and SEM. All four models employ the same scheduling algorithm used in MISE and allocate equal memory bandwidth to each application. Thus, STFM, MISE, and SEM have the same system performance.



(a) mcf (b) libquantum (c) sphinx3

(d) cactusADM (e) sjeng (f) gcc

Fig. 2. Comparison of IPC with the request service rate.

## 6 EVALUATION RESULTS

### 6.1 IPC Versus the Request Service Rate

To compare IPC to the request service rate, we implement a modified version of MISE, called *MISEIPC*. MISEIPC replaces the number of requests served by the number of committed instructions and calculates the slowdown of each application by Equation (2). The other parts of MISEIPC and MISE are the same. Figure 2 compares the accuracy of MISEIPC to MISE for six representative applications. The six representative applications are selected by varying memory intensity: *gcc* and *sjeng* are memory nonintensive, and the others are memory intensive. Each application is run on a four-core, one-channel system along with three other applications: *lbm*, *zeusmp*, and *milc*. We can make two observations. First, the accuracy of MISEIPC for memory-intensive applications is a bit lower than that of MISE. The estimation error of $mcf$, *libquantum*, *sphinx*3, and *cactusADM* in MISEIPC is 3.79%, 1.70%, 27.84%, and 11.63%, respectively, whereas that in MISE is 2.96%, 1.66%, 30.47%, and 10.33%, respectively. Second, the accuracy of MISEIPC for memory-nonintensive applications is much higher than that of MISE. In MISEIPC, the estimation error of *gcc* and *sjeng* is 2.49% and 7.15%, respectively, whereas the estimation error of *gcc* and *sjeng* in MISE is 11.62% and 12.80%, respectively. This mainly occurs for two reasons: (1) it is hard to accurately estimate the small ARSR for memory-nonintensive applications, and (2) the model to compute the slowdown of memory-nonintensive applications is inaccurate, although MISE takes into account the duration of the compute phase. MISEIPC has no such problems, as IPC contains all information of the compute and memory phase. Therefore, compared to the request service rate, IPC can unify the slowdown estimation model, improve the estimation accuracy for memory-nonintensive applications, and keep the estimation accuracy for memory-intensive applications.

### 6.2 Comparison With STFM and MISE at Shared Memory

Figure 3 compares the average slowdown estimation error from STFM, MISE, MISEIPC, and SEM. We can draw three conclusions. First, STFM has the highest estimation error for nearly all applications, although STFM considers bank-level parallelism, row-buffer, and data bus interference. The average estimation error of STFM is up to 30.15%, which is much higher than that of MISE, MISEIPC, and SEM. There are two reasons for low estimation accuracy of STFM: (1) it is not right

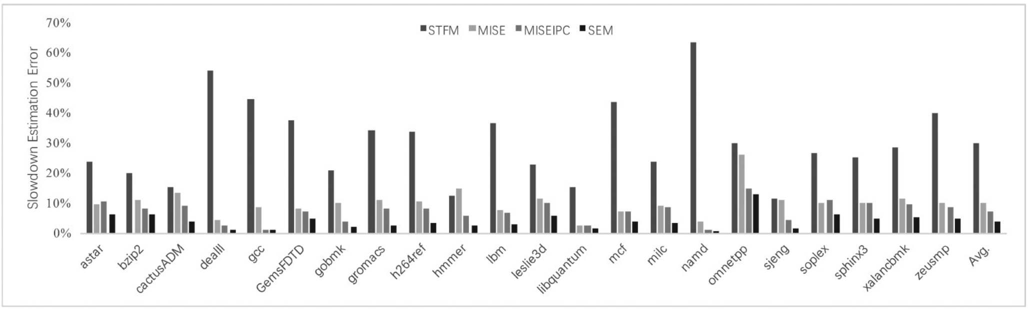


Fig. 3. Comparison of SEM with STFM, MISE, and MISEIPC.

to estimate an application's slowdown as the ratio of its memory stall time when run alone versus when running alongside other applications, as the stall time only considers progress in the memory phase, and for a compute-intensive application, the stall time on memory has little influence on the application's performance, and STFM significantly overestimates the application's slowdown (as shown in Figure 3, gcc, namd, and dealII are typical compute-intensive applications, and their estimation error is up to 44.66%, 63.65%, and 54.21%, respectively); (2) STFM estimates the number of interference cycles for an application while the application is receiving significant interference from other applications, and at the existence of abundant memory-level parallelism, it is very difficult to determine the number of cycles by which each request is delayed due to memory interference, thus the estimation error of memory-intensive applications is also very high in STFM.

Second, the average estimation error of MISE decreases from 10.1% to 7.28% when the request service rate is replaced by IPC. The improved estimation accuracy comes from the accurate slowdown estimation for compute-intensive applications. For example, the estimation error of gcc is decreased from 8.7% to 1.27%, and the estimation error of *gobmk* is decreased from 10.1% to 3.88%. Although MISE takes the compute phase into account by estimating the stall fraction, the slowdown estimation model for compute-intensive applications in MISE is not accurate. As shown in Figure 2(e), the stall fraction of sjeng fluctuates around the stall fraction threshold. When the stall fraction of sjeng is below the stall fraction threshold, the slowdown of sjeng is significantly underestimated. For memory-intensive applications, the estimation accuracy of MISEIPC is almost the same as that of MISE. Memory-intensive applications spend a large fraction of their execution time on memory accesses, so the request service rate of a memory-intensive application has significant impact on its performance. MISE estimates the ARSR of an application when the application is given the highest priority, so the estimated ARSR is more accurate since the interference received by the application is minimized. Therefore, MISE makes a marked progress in slowdown estimation accuracy.

Third, SEM decreases the average estimation error to 4.06%. SEM uses the framework of MISE to reduce memory interference when estimating alone performance of an application. SEM also measures the performance IPC directly and unifies the slowdown estimation model for both compute-intensive and memory-intensive applications. There are two key differences between MISE and SEM. First, MISE only considers the interference during read drain mode, whereas SEM considers the interference during both read drain and write drain modes. In write drain mode, all read requests in the request buffer are blocked. If the highest-priority application was run alone, its read requests would be blocked by its write requests. But in a shared system, the write queue stores the write requests from all applications, and write requests from other applications may be served

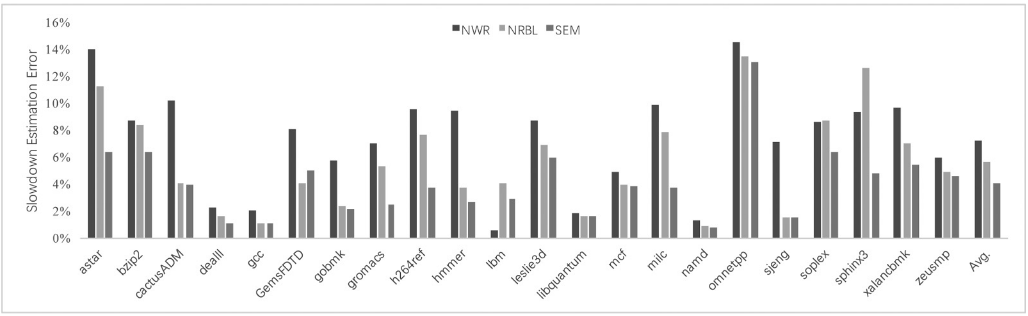


Fig. 4. Effect of write and row-buffer interference.

before the write drain mode is disabled. For an application that has no write requests, the cycles during which read requests from the application are blocked by write requests from other applications are interference cycles. Second, MISE uses a simple mechanism to determine the number of interference cycles, whereas SEM considers bank-level parallelism, and row-buffer and data bus interference by the per-bank structure. Consider one cycle when an activate command from the highest-priority application is waiting in bank 0 and a precharge command from other applications in bank 1 is scheduled. The scheduled precharge command does not cause any timing constraints on the activate command in bank 0. MISE treats this cycle as an interference cycle, whereas SEM thinks that interference may not happen in this cycle. MISE easily overestimates memory interference in read drain mode, and the overestimation makes up the dismissal of write interference to some extent. Therefore, SEM can provide better accuracy than STFM, MISE, and MISEIPC in application slowdown estimation.

## 6.3 Effect of Write and Row-Buffer Interference

Figure 4 shows the estimation error of each application when SEM does not consider write interference or row-buffer interference. We can make two observations. First, the average estimation error of NWR is 7.78%, close to the average estimation error of MISEIPC at 7.28%. When write interference is not considered, the estimation error of most applications increases significantly. For example, the estimation error increases from 3.96% to 9.97% for cactusADM, from 1.58% to 7.21% for sjeng, and from 3.92% to 5.03% for mcf. Second, the average estimation error of NRBL is 5.58%. The effect of row-buffer interference on estimation accuracy is weaker than that of write interference. For applications with low row-buffer locality, ignoring row-buffer interference has little influence on estimation accuracy, such as cactusADM, mcf, and sjeng. The read page hit rate of cactusADM is only 6.2%. The estimation error of libquantum remains the same when row-buffer interference is not considered. This is because libquantum has a very high page hit rate and high memory intensity simultaneously. Requests from other applications have no chance to close the rows opened by libquantum. For applications that have less memory intensity and high row-buffer locality, the estimation error increases significantly when SEM does not take row-buffer interference into account, such as from 3.59% to 7.45% for h264ref, from 3.55% to 7.80% for milc, and from 4.83% to 12.77% for sphinx3. The estimation error of omnetpp is very high for NWR, NRBL, and SEM, greater than 12.0%. This is because the MPKI of omnetpp varies significantly with time. The average alone performance in highest-priority epochs may be not close to the real alone performance in a quantum. Hence, we can conclude that SEM can improve slowdown estimation accuracy a lot by taking write and row-buffer interference into account.

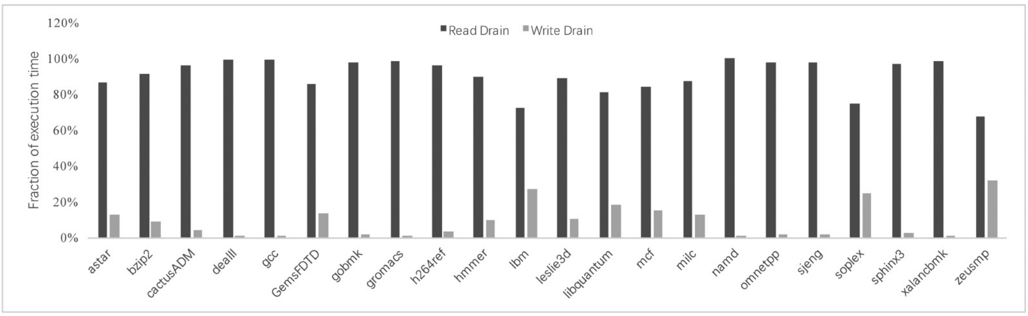


Fig. 5. Fraction of read drain and write drain cycles.

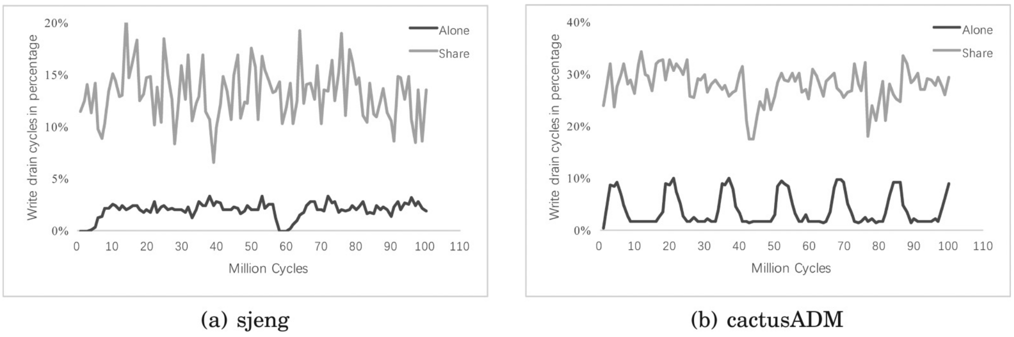


Fig. 6. Fraction of write drain cycles in alone and shared systems.

Figure 5 shows the fraction of read drain and write drain cycles for all applications when they are running alone. Most applications spend less than 10.0% of execution time in write drain mode, such as 4.02% for cactusADM, 2.01% for sjeng, and 3.65% for h264ref. Some memory-intensive applications spend a large fraction of execution time in write drain mode, such as 32.27% for zeusmp, 15.43% for mcf, and 25.20% for soplex. When sjeng is running along with soplex, read requests from sjeng will be blocked by write requests from soplex in write drain mode. Figure 6 shows the fraction of highest-priority cycles on write drain for sjeng and cactusADM in the alone and shared systems. We only track the write drain cycles when there are waiting read requests from the highest-priority application in the request buffer. In the shared system, sjeng and cactusADM are run along with *lbm*, *zeusmp*, and *milc*. As we expected, the length of the write drain cycles when sjeng or cactusADM is running with other applications is much higher than that if they had run alone. The difference between the two situations is the write interference cycles when read requests from the highest-priority application are blocked by write requests from other applications. The ratio of write interference cycles and highest-priority cycles is about 10.0% for sjeng and 20% for cactusADM. Thus, it is necessary to consider write interference when estimating alone performance.

Figure 7 shows the estimated slowdown of NWR and SEM for sjeng and cactusADM, which are are run along with *lbm*, *zeusmp*, and *milc*. Therefore, MISE and SEM can be compared by combining Figures 2 and 7. We can make two observations. First, the estimation error of NWR is very high: 21.14% for cactusADM and 10.85% for sjeng. These two values are close to the difference between alone and shared systems in Figure 5. Since sjeng and cactusADM have very low row-buffer locality, the effect of row-buffer interference can be ignored. The difference between NWR and MISEIPC

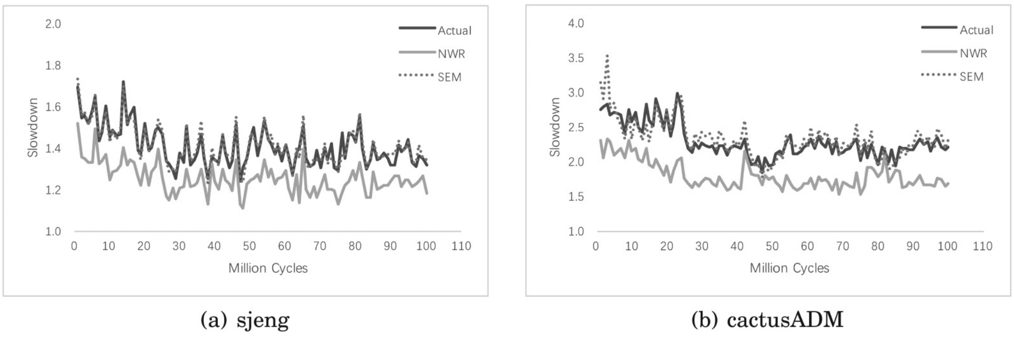


Fig. 7. Comparison of SEM with NWR for representative applications.

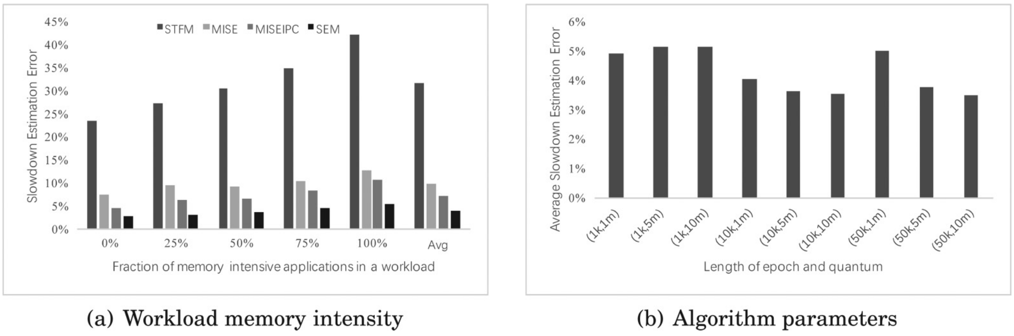
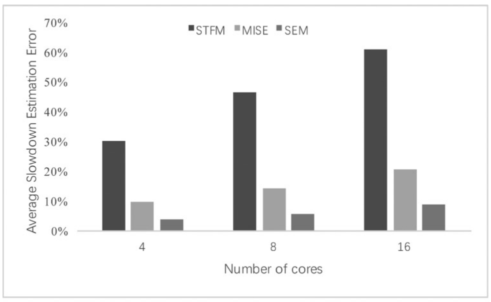


Fig. 8. Sensitivity to workload intensity and algorithm parameters.

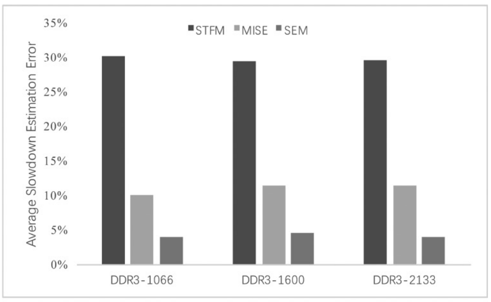
is the structure to identify interference cycles. MISEIPC uses a per-channel counter to store the application id of a previously scheduled command, whereas NWR uses a per-channel counter to monitor data bus interference and multiple per-bank counters to monitor intrabank interference. NWR also considers the timing constraints caused by previously scheduled commands from other applications. Thus, the simple mechanism in MISEIPC overestimates the number of interference cycles in read drain mode. The result is that MISEIPC has lower estimation error than NWR for sjeng and cactusADM. Second, the estimation error of SEM is very low: 1.61% for sjeng and 4.79% for cactusADM. Sometimes SEM may overestimate the slowdown of cactusADM because it is difficult to estimate the accurate value of write drain cycles. In a shared system, write drain mode is enabled more frequently. The bank-level parallelism of the highest-priority application may be broken in write drain mode because there are few write requests from the highest-priority application in the write queue. If the application had run alone, the write queue would be filled with write requests from the highest-priority application. However, SEM still provides higher accuracy than STFM and MISE.

## 6.4 Sensitivity to Memory Intensity and Algorithm Parameters

Figure 8(a) shows the slowdown estimation error of STFM, MISE, MISEIPC, and SEM by varying workload memory intensity from 0% to 100%. We can draw two conclusions. First, SEM outperforms STFM, MISE, and MISEIPC in terms of slowdown estimation accuracy across all memory-intensity categories. Second, the slowdown estimation error tends to increase with workload memory intensity, but not absolutely. For MISE, the estimation error in the 25% workload



(a) Number of cores



(b) DRAM frequency

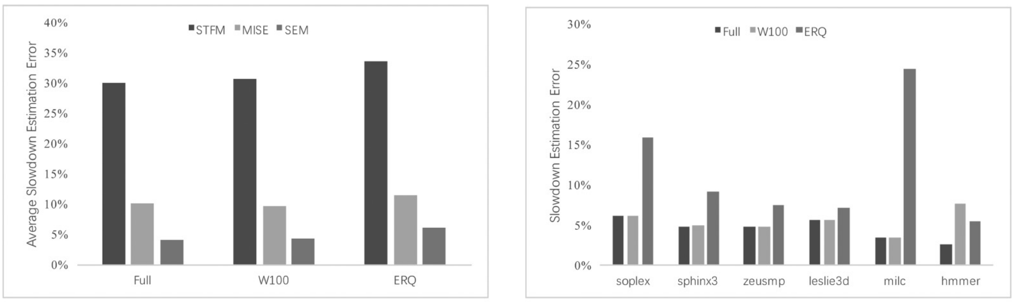
Fig. 9. Sensitivity to cores and DRAM frequency.

memory-intensity category is 9.65%, which is higher than that in the 50% workload memory-intensity category (9.23%). This is because MISE has low slowdown estimation accuracy for memory-nonintensive applications. For SEM, the estimation error increases slightly with workload memory intensity. In 100% workload memory intensity, the estimation error is only 5.6%. All slowdown estimation models, including SEM, only consider the major parts of memory interference now because it is difficult to accurately quantify memory interference during interference cycles. Therefore, when an application is interfered with heavily by other applications for a long time, the estimation error of this application tends to be large.

Figure 8(b) shows the average slowdown estimation error of SEM by varying the length of quantum and epoch. We can make three observations. First, when the length of epoch is small (1 kilocycles), the estimation error is very high. Due to the timing constraints and long latency of DRAM, the requests from the highest-priority application cannot be scheduled immediately. It needs some time to close the rows opened by other applications. The ratio of these waiting cycles and the epoch length is pretty high when the epoch length is very small. Second, when the length of epoch is 50 kilocycles and the length of quantum is 1 million cycles, the estimation error is very high because the number of epochs in a quantum reduces a lot. The number of epochs assigned to an application may be very low, even zero, preventing the estimation of its alone performance. When few epochs are assigned to an application, the alone performance during these epochs may be not close to the alone performance in a quantum, especially when the phase behavior of this application changes. Third, when the epoch length is over 10 kilocycles, the average estimation error tends to decrease with the ratio of quantum length and epoch length. When the length of quantum is larger, the performance is more stable and each application has more highest-priority epochs, so the alone performance of each application can be estimated effectively. For our evaluations, we use a quantum of 1 million cycles and an epoch of 10 kilocycles.

## 6.5 Sensitivity to Cores and DRAM Frequency

Figure 9(a) shows the estimation error of STFM, MISE, and SEM by varying the number of cores in a one-channel system. First, the estimation error of all models increases with the number of cores. When the number of cores increases, the system becomes more bandwidth constrained and the interapplication interference are extremely heavy. Since the number of epochs in a quantum is fixed, each application has fewer highest-priority epochs when the number of cores increases. The accuracy of estimated alone performance in MISE and SEM primarily depends on the number of highest-priority epochs. This, the average estimation error of MISE and SEM increases with the core counts. Second, the estimation error of SEM is much smaller than that of MISE in all system



(a) SEM vs. STFM and MISE (b) Estimation accuracy for some applications

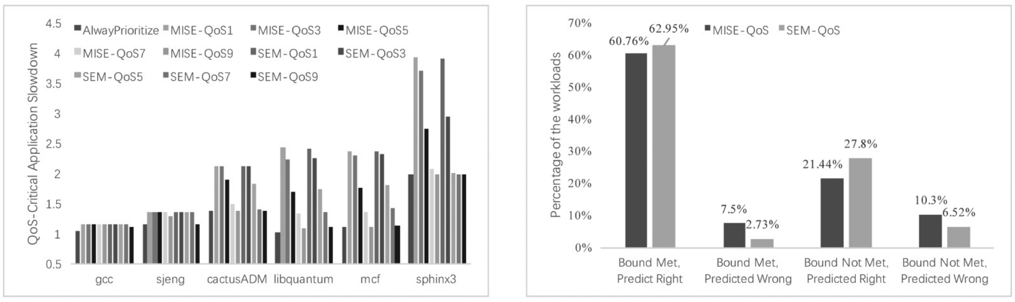
Fig. 10. Sensitivity to the write drain policy.

configurations. On the 16-core, one-channel system, the estimation error of MISE is up to 21.03%, whereas the estimation error of SEM is only 8.9%. Therefore, SEM can provide high accuracy in application slowdown on different system configurations.

Figure 9(b) shows the estimation error of STFM, MISE, and SEM by varying the frequency of DRAM. The frequency of the processor is fixed at 2.133GHz, and we only change the ratio of processor clock and DRAM clock: 4:1 for DDR3-1066, 3:1 for DDR3-1600, and 2:1 for DDR3-2133. We can observe that the estimation error of STFM, MISE, and SEM changes slightly when the frequency of DRAM increases. In DDR3-1600, the estimation error of STFM, MISE, and SEM is 29.42%, 11.43%, and 4.51%, respectively. In DDR3-2133, the estimation error of STFM, MISE, and SEM is 29.56%, 11.41%, and 4.06%, respectively. For memory-nonintensive applications, the estimation error of STFM increases with DRAM frequency, whereas that of MISE and SEM decreases because each request has shorter latency and the fraction of execution time in the compute phase increases. For memory-intensive applications, the increased DRAM frequency weakens their ability to cause memory interference. In DDR3-1066, libquantum can occupy memory bandwidth for a long time and receive little memory interference, so MISE and SEM have low estimation error for libquantum: 2.53% and 1.63%, respectively, whereas in DDR3-2133, since the row-hit streams of libquantum are easily broken by requests from other applications, libquantum receives heavy memory interference. The estimation error for libquantum is 21.03% in MISE and 2.95% in SEM. Therefore, SEM can provide high accuracy in application slowdown for different DRAM models.

## 6.6 Sensitivity to the Write Drain Policy

Figure 10(a) shows the average estimation error of STFM, MISE, and SEM by varying the write drain policy. *Full* means the write drain policy on the baseline system, and ERQ means that write drain mode is enabled when the write queue length is over the high watermark or the read queue is empty. *W100* means that write drain mode is enabled when the write queue length is over the high watermark or the read queue is empty for more than 100 processor cycles. Full is equal to Winf (the read queue is empty for infinite cycles), and ERQ is equal to W0 (the read queue is empty for 0 cycles). We can draw two conclusions. First, the estimation accuracy of all models increases with the waiting cycles. For the ERQ policy, if an application has been run alone, its write requests can be served when the read queue is empty and does not block its read requests. But in the shared system, the memory controller schedules read requests from other applications when there are no read requests from the highest-priority application in the read queue. The unscheduled write requests may block read requests from the highest-priority application when the write queue length is over the high watermark. Therefore, it is more difficult to accurately quantify the write interference in



(a) QoS-Critical application performance (b) QoS effectiveness

Fig. 11. Comparison of SEM-QoS with MISE-QoS.

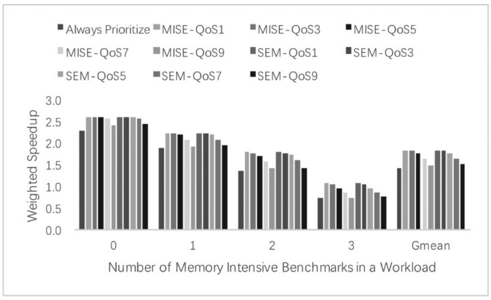
ERQ. Second, when the write drain policy is ERQ, the average estimation error of SEM is only 6.1%, which is much lower than that of MISE (11.45%). For most applications, the estimation error changes a little with different write drain policies. Figure 10(b) shows some applications whose estimation error increases significantly in ERQ. A common feature of these applications is that the applications drain most of the write requests when the read queue is empty. The fraction of execution time on the write drain when the read queue is empty is 15.5% for soplex, 9.9% for sphinx3, 9.14% for zeusmp, 18.0% for leslie3d, 26.65% for milc, and 34.35% for hmmer. Currently, we have no good idea for estimating these cycles and leave it as future work. Therefore, we can conclude that SEM provides higher slowdown estimation accuracy than STFM and MISE when the commonly used write drain policies are employed.

## 6.7 Comparison of SEM-QoS with MISE-QoS

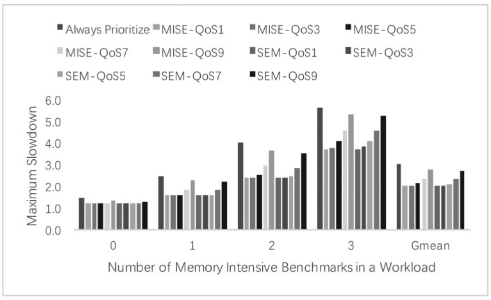
We use the QoS mechanism in MISE to provide QoS for a critical application. The QoS-critical application is given the total memory bandwidth initially, and its slowdown is estimated at the end of each quantum. If the estimated slowdown is over the QoS bound, the controller reduces the bandwidth allocated to the QoS-critical application by 2%. If the estimated slowdown is less than the QoS bound, the controller also increases the bandwidth allocated to the QoS-critical application by 2%. The left bandwidth is allocated equally to other applications. Here, the QoS bound is set to the value $10/n$, where $n$ is the QoS value. After 50 quanta, the allocated bandwidth for the QoS-critical application is relatively stable. In later quanta, we conclude that the QoS bound cannot be met if the QoS-critical application does not meet the QoS bound for the consecutive 5 quanta (empirical value). We run each application with 12 fixed three-core workloads.

Figure 11(a) shows the performance for six representative QoS-critical applications by varying the QoS value. Each application is run along with *lbm*, *zeusmp*, and *milc*. We can make three observations. First, the slowdown of most applications is considerably more than one when the application is always prioritized. Second, the slowdown of QoS-critical applications in MISE-QoS and SEM-QoS gradually decreases with the QoS value. When the QoS value increases, MISE and SEM allocate more bandwidth to the QoS-critical application. Third, compared to MISE-QoS, SEM-QoS is more sensitive to the QoS value. When the QoS-value is 9, gcc in MISE-QoS has a slowdown of 1.1625, over the QoS bound 1.11, whereas gcc in SEM-QoS has a slowdown of 1.1055, below the QoS bound. This is because MISE significantly underestimates the slowdown of gcc.

Figure 11(b) shows the effectiveness of MISE-QoS and SEM-QoS across 1,320 workloads. We can find that SEM-QoS is more effective than MISE-QoS to correctly predict whether the QoS bound is met or not. For 62.95% of the workloads, SEM-QoS can meet the specified bound and correctly



(a) System Performance



(b) Fairness

Fig. 12. System performance and fairness of non–QoS-critical applications across 1,320 workloads.

predict that the bound is met. SEM-QoS cannot meet the specified bound but correctly predicts that the bound is not met for 27.8% of workloads, whereas the value for MISE-QoS is only 21.44%. When the slowdown of the QoS-critical application is significantly overestimated, the application can meet the specified bound, but the slowdown estimation model predicts that the bound is not met. However, when the slowdown of the QoS-critical application is significantly underestimated, the application cannot meet the specified bound, but the slowdown estimation model predicts that the bound can be met. Compared to MISE-QoS, SEM-QoS has less fraction of the workloads where the QoS-critical application is significantly overestimated or underestimated. Therefore, we can conclude that SEM is more robust to provide QoS for a single application than MISE.

Figure 12 compares the weighted speedup and maximum slowdown of AlwaysPrioritize, MISE-QoS, and SEM-QoS for non–QoS-critical applications across 1,320 workloads. We can make three conclusions. First, compared to AlwaysPrioritize, the benefits of MISE-QoS and SEM-QoS for system performance and fairness increase with memory intensity in a workload because strictly prioritizing a memory-intensive application will cause significant interference to other applications. When there are three memory-intensive applications in a workload, MISE-QoS5 and SEM-QoS5 improve system performance by 33% and fairness by 27% compared to AlwaysPrioritize. Second, when the QoS value increases, the system performance and fairness of MISE-QoS and SEM-QoS approach that of AlwaysPrioritize. Finally, the benefits of SEM-QoS over MISE-QoS for system performance and fairness increase with the QoS value. SEM-QoS9 has 1.57% higher weighted speedup and 2.44% lower maximum slowdown than MISE-QoS9. Therefore, SEM-QoS can achieve high system performance and fairness across non–QoS-critical applications.

## 6.8 IPC Versus the Cache Access Rate

To compare IPC to the cache access rate, we add a shared cache before main memory and implement ASM to estimate application slowdown. In ASM, MISE estimates the queueing cycles for normal cache misses at main memory. Figure 13 shows the slowdown for six representative applications of ASM and ASMIPC. The difference between ASMIPC and ASM is that cache access rate is replaced by IPC. The result is similar to the comparison of IPC with the request service rate. For applications with a high cache access rate, the estimation error of ASMIPC is similar to that of ASM. For mcf, the estimation error is 14.55% in ASM and 15.09% in ASMIPC. For libquantum, the estimation error is 28.55% in ASM and 28.57% in ASMIPC. For applications with a low cache access rate, ASMIPC has higher estimation accuracy than ASM. For sjeng, the estimation error is 6.53% in ASM and only 1.97% in ASMIPC. For gcc, the estimation error is 11.26% in ASM and only 1.64% in ASMIPC. This is because these applications spend a large fraction of execution time in

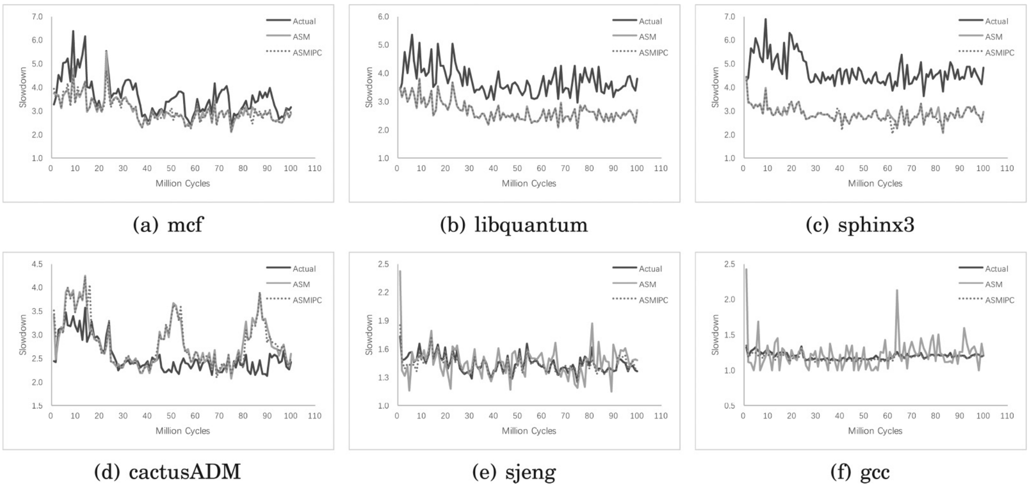


Fig. 13. Comparison of IPC with the cache access rate.

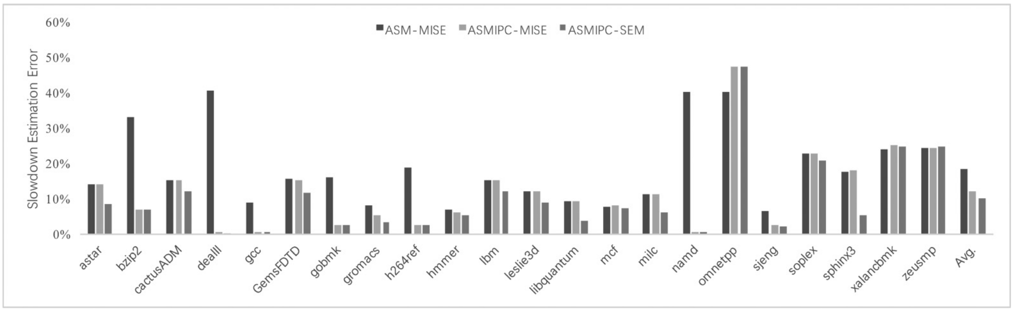


Fig. 14. Comparison of SEM with MISE at shared cache.

the compute phase, and the cache access rate only represents the performance in the cache phase. Hence, IPC is more effective than the cache access rate in slowdown estimation, especially for compute-intensive applications.

Figure 14 shows the estimation error of ASM-MISE, ASMIPC-MISE, and ASMIPC-SEM across 80 four-core workloads. The word after ASM means the model to estimate queueing cycles in the memory controller. We can make three observations. First, for compute-intensive applications, ASMIPC-MISE has much higher estimation accuracy than ASM-MISE. For example, the estimation error of dealII decreases from 40.51% to 0.76%. Second, compared to ASMIPC-MISE, ASMIPC-SEM can provide higher estimation accuracy for most applications. For example, the estimation error of astar decreases from 14.09% to 8.64% because MISE overestimates the queueing cycles of astar. Third, the average estimation error of ASMIPC-SEM is 10.04%, which is smaller than that of ASM-MISE (18.73%). The estimation error of some applications is very high, and we leave this problem as future work. Hence, we can conclude that SEM is better than MISE even in the system with a shared cache.

## 7 CONCLUSIONS

Most previous work has focused on alleviating memory interference, with the goal of high system performance and fairness. Few prior works have tried to quantify memory interference and provide predictable performance, but the estimation accuracy is relatively low. Thus, we propose a new slowdown estimation model at main memory called *SEM*. SEM unifies the slowdown estimation model for different applications by IPC. Compared to measuring the cache access rate or requests service rate, measuring IPC directly can significantly improve the estimation accuracy for compute-intensive applications without degrading the estimation accuracy for memory-intensive or cache-sensitive applications. To estimate the alone performance of an application, SEM assigns the application the highest priority to minimize memory interference. To quantify the number of interference cycles, SEM uses the per-bank structure to monitor interference and considers write interference, row-buffer interference, and data bus interference. We compare SEM to STFM and MISE on the baseline system. The estimation error of STFM and MISE is 30.15% and 10.10%, respectively, whereas the estimation error of SEM is only 4.06%. We also compare SEM to MISE on the system with a shared cache. The results show that SEM has much higher accuracy than STFM and MISE in application slowdown estimation. We can conclude that SEM is a promising substrate to provide predictable performance in modern and future multicore systems.

## ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable feedback, which immensely helped improve the draft.

## REFERENCES

Kevin Kai-Wei Chang, Donghyuk Lee, Zeshan Chishti, Alaa R. Alameldeen, Chris Wiklerson, Yoongu Kim, and Onur Mutlu. 2014. Improving DRAM performance by parallelizing refreshes with accesses. In *Proceedings of HPCA 2014*.

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 http://www.cs.utah.edu/~rajeev/jwac12/.

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 HPCA 2013*.

Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R. Das. 2009. Application-aware prioritization mechanisms for on-chip networks. In *Proceedings of MICRO 2009*.

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 ASPLOS 2010*.

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 MICRO 2011*.

David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. 2012. Bandwidth bandit: Quantitative characterization of memory contention. In *Proceedings of PACT 2012*.

Stijn Eyerman and Lieven Eeckhout. 2009. Per-thread cycle accounting in SMT processors. In *Proceedings of ASPLOS 2009*.

Saugata Ghose, Hyodong Lee, and José F. Martínez. 2013. Improving memory scheduling via processor-side load criticality information. In *Proceedings of ISCA 2013*.

John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. In *Proceedings of CAN 2006*.

Ibrahim Hur and Calvin Lin. 2004. Adaptive history-based memory schedulers. In *Proceedings of MICRO 2004*.

Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of ISCA 2008*.

Canturk Isci and Margaret Martonosi. 2003. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of MICRO 2003*.

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 DAC 2012*.

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 HPCA 2012*.

Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. 2011. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *Proceedings of MICRO 2011*.

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 HPCA 2014*.

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 RTAS 2014*.

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. In *Proceedings of RTAS 2014*.

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 HPCA 2010*.

Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010b. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of MICRO 2010*.

Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A fast and extensible DRAM simulator. In *Proceedings of CAL 2015*.

Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. 2008. Prefetch-aware DRAM controllers. In *Proceedings of MICRO 2008*.

Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2010. *DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems*. Technical Report. Carnegie Institute of Technology, Carnegie Mellon University, Pittsburgh, PA.

Junghoon Lee, Hanjoon Kim, Minjeong Shin, John Kim, and Jaehyuk Huh. 2014. Mutually aware prefetcher and on-chip network designs for multi-cores. In *Proceedings of TC 2014*.

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 PACT 2012*.

Peng Liu, JiYang Yu, and Michael C. Huang. 2016. Thread-aware adaptive prefetcher on multicore systems: Improving the performance for multithreaded workloads. *ACM Transactions on Architecture and Code Optimization* 13, 1, Article No. 13.

Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of MICRO 2011*.

Asit K. Mishra, Onur Mutlu, and Chita R. Das. 2013. A heterogeneous multiple network-on-chip design: An application-aware approach. In *Proceedings of DAC 2013*.

Janani Mukundan and José F. Martínez. 2012. MORSE: Multi-objective reconfigurable self-optimizing memory scheduler. In *Proceedings of HPCA 2012*.

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 MICRO 2011*.

Onur Mutlu. 2013. Memory scaling: A systems architecture perspective. In *Proceedings of IMW 2013*.

Onur Mutlu, Justin Meza, and Lavanya Subramanian. 2015. The main memory system: Challenges and opportunities. In *Proceedings of IISE 2015*.

Onur Mutlu and Thomas Moscibroda. 2007. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of MICRO 2007*.

Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proceedings of ISCA 2008*.

Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut Kandemir, Anand Sivasubramaniam, Onur Mutlu, and Chita R. Das. 2012. Application-aware prefetch prioritization in on-chip networks. In *Proceedings of PACT 2012*.

Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory access scheduling. In *Proceedings of ISCA 2000*.

Vivek Seshadri, Abhishek Bhowmick, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2014. The dirty-block index. In *Proceedings of ISCA 2014*.

Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *Proceedings of ASPLOS 2002*.

Jeffrey Stuecheli, Dmimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. 2010. The virtual write queue: Coordinating DRAM and last-level cache policies. In *Proceedings of ISCA 2010*.

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 ICCD 2014*.

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 MICRO 2015*.

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 HPCA 2013*.

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.

Xiyue Xiang, Saugata Ghose, Onur Mutlu, and Nian-Feng Tzeng. 2016. A model for application slowdown estimation in on-chip networks and its use for improving system fairness and performance. In *Proceedings of ICCD 2016.*

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 HPCA 2014.*

Dongliang Xiong, Kai Huang, Xiaowen Jiang, and Xiaolang Yan. 2016. Memory access scheduling based on dynamic multi-level priority in shared DRAM systems. *ACM Transactions on Architecture and Code Optimization* 13, 4, Article No. 42.

Received December 2016; revised May 2017; accepted June 2017