MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems

Lavanya Subramanian Vivek Seshadri Yoongu Kim Ben Jaiyen Onur Mutlu
Carnegie Mellon University

Abstract

Applications running concurrently on a multicore system interfere with each other at the main memory. This interference can slow down different applications differently. Accurately estimating the slowdown of each application in such a system can enable mechanisms that can enforce quality-of-service. While much prior work has focused on mitigating the performance degradation due to inter-application interference, there is little work on estimating slowdown of individual applications in a multi-programmed environment. Our goal in this work is to build such an estimation scheme.

To this end, we present our simple Memory-Interference-induced Slowdown Estimation (MISE) model that estimates slowdowns caused by memory interference. We build our model based on two observations. First, the performance of a memory-bound application is roughly proportional to the rate at which its memory requests are served, suggesting that request-service-rate can be used as a proxy for performance. Second, when an application's requests are prioritized over all other applications' requests, the application experiences very little interference from other applications. This provides a means for estimating the uninterfered request-service-rate of an application while it is run alongside other applications. Using the above observations, our model estimates the slowdown of an application as the ratio of its uninterfered and interfered request service rates. We propose simple changes to the above model to estimate the slowdown of non-memory-bound applications.

We demonstrate the effectiveness of our model by developing two new memory scheduling schemes: 1) one that provides soft quality-of-service guarantees and 2) another that explicitly attempts to minimize maximum slowdown (i.e., unfairness) in the system. Evaluations show that our techniques perform significantly better than state-of-the-art memory scheduling approaches to address the above problems.

1. Introduction

Main memory is a critical shared resource in modern multicore systems. Multiple applications running concurrently on a multicore system contend with each other for the available memory bandwidth. This inter-application interference degrades both individual application and overall system performance. Moreover, the slowdown of each application depends on the other concurrently running applications and the available memory bandwidth. Hence, different applications experience different and unpredictable slowdowns, as demonstrated by previous work [26, 28, 29, 30]. Accurately estimating this memory-interference induced slowdown can enable mechanisms to better enforce Quality of Service (QoS) and fairness in multicore systems. For example, the memory controller could use accurate slowdown estimates to manage memory bandwidth appropriately to provide soft QoS guarantees to applications. Alternatively, conveying the slowdown of each application to the operating system (OS) may allow the OS to make better application scheduling decisions – e.g., the OS can co-schedule applications that interfere less with each other.

A considerable number of prior works have proposed several different approaches to mitigate interference between applications at the main memory, with the goal of improving overall system performance and/or fairness. Examples of such approaches include new memory scheduling (e.g., [2, 19, 20, 28, 29, 30]), memory channel/bank partitioning [16, 27], memory interleaving [18], and source throttling [6] techniques.

Although these previous proposals are effective in mitigating the performance degradation due to memory interference, few of them, e.g., [5, 6, 28] attempt to estimate the actual slowdown of each application compared to when the application is run alone. In this work, we find that the prior approaches [6, 28] to estimate individual application slowdown due to main memory interference are inaccurate. Part of the reason for their inaccuracy is that these mechanisms were not designed to accurately estimate slowdown of individual applications, with the goal of providing predictable performance. Rather, they use the estimated slowdown information to make informed prioritization, throttling or scheduling decisions to improve system performance and fairness. While system performance and fairness continue to be important considerations, the ability to achieve predictable performance for different applications is gaining importance in today's era of workload consolidation and concurrent execution of multiple applications, as exemplified by server consolidation where different users' jobs that are consolidated onto the same machine share resources.

Our goal in this work is to provide predictable performance for individual applications. To this end, we design a model to accurately estimate memory-interference-induced slowdowns of individual applications running concurrently on a multicore system. Estimating the slowdown of an individual application requires two pieces of information: 1) the performance of the application when it is run concurrently with other applications, and 2) the performance of the application when it is run alone on the same system. While the former can be directly measured, the key challenge is to estimate the performance the application would have if it were running alone *while* it is actually running alongside other applications. This requires quantifying the effect of interference on performance. In this work, we make two observations that lead to a simple and effective mechanism to estimate the slowdown of individual applications.

Our first observation is that performance of a memory-bound application is roughly proportional to the rate at which its memory requests are served. This observation suggests that we can use request-service-rate as a proxy for performance, for memory-bound applications. As a result, slowdown of such an application can be computed as the ratio of the request-service-rate when the application is run alone on a system (*alone-request-service-rate*) to that when it is run alongside other interfering applications (*shared-request-service-rate*). Although the *shared-request-service-rate* can be measured in a straightforward manner using a few counters at the memory controller, this model still needs to estimate the *alone-request-service-rate* of an application *while* it is run alongside other applications.

Our second observation is that the *alone-request-service-rate* of an application can be estimated by giving the application's

requests the highest priority in accessing memory. Giving an application's requests the highest priority in accessing memory results in very little interference from other applications' requests. As a result, most of the application's requests are served as though the application has all the memory bandwidth for itself, allowing the system to gather a good estimate for the *alone-request-service-rate* of the application. We make our model more accurate by accounting for the little interference caused by other applications' requests due to queuing delays.

Based on the above two observations, our proposed Memory-Interference-induced Slowdown Estimation (MISE) model works as follows. The memory controller assigns priorities to applications such that every application executing on the system periodically receives the highest priority to access memory. Every time an application receives the highest priority, the memory controller estimates the application's *alone-request-service-rate*. This *alone-request-service-rate* estimate along with the measured *shared-request-service-rate* can be used to estimate the slowdown of an application.

Although the above model works well for memory-bound applications, we find that it is not accurate for non-memory-bound applications. This is because a non-memory-bound application spends a significant fraction of its execution time in the *compute phase*, in which the core does not stall waiting for a memory request. As a result, request-service-rate cannot be used as a direct proxy for performance of such applications. Therefore, to make our MISE model accurate for non-memory-bound applications as well, we augment it to take into account the duration of the compute phase. Section 3 provides more details of our MISE model.

Our slowdown estimation model can enable several mechanisms to provide QoS guarantees and achieve better fairness. We build two new memory scheduling mechanisms on top of our proposed model to demonstrate its effectiveness.

The first mechanism, called MISE-QoS, provides soft QoS guarantees for applications of interest while trying to maximize the performance of all other applications in a best-effort manner. The memory controller ensures that the applications of interest meet their slowdown requirements by allocating them just enough memory bandwidth, while scheduling the requests of other applications in a best-effort manner to improve overall system performance. First, we show that when there is one application of interest, MISE-QoS meets the target slowdown bound for 80.9% of our 3000 tested data points, while significantly improving overall system performance compared to a state-of-the-art approach that always prioritizes the requests of the application of interest [15]. Furthermore, MISE-QoS correctly predicts whether or not the bound was met for 95.7% of data points, whereas the state-of-the-art approach [15] has no provision to predict whether or not the bound was met. Next, we demonstrate that even when there are multiple applications of interest, MISE-QoS can meet the target slowdown bound for all applications of interest, while still providing significant system performance improvement.

The second mechanism, called MISE-Fair, attempts to minimize the maximum slowdown (i.e., unfairness [4, 19, 20]) across all applications. It does so by estimating the slowdown of each application and redistributing bandwidth to reduce the slowdown of the most slowed-down applications. We show that our approach leads to better fairness than three state-of-the-art application-aware memory access scheduling approaches [19, 20, 28]. In this use case, the memory controller can also potentially convey the achievable maximum slowdown to the operating system (OS). The OS can in turn use this information to make better scheduling decisions.

Our paper makes the following contributions:

- We propose a new model for estimating the slowdown of individual applications concurrently running on a multi-core system. Implementing our model requires only simple changes to the memory controller hardware.
- We compare the accuracy of our model to a previously proposed slowdown estimation model, Stall-Time Fair Memory scheduling (STFM) [28], and show that our model is significantly more accurate than STFM's model.
- We show the effectiveness of our model by building and evaluating two new memory bandwidth management schemes on top of it, one that provides soft QoS guarantees, while optimizing for performance in a best-effort manner (MISE-QoS) and another that improves overall system fairness (MISE-Fair). Our proposed approaches perform better than state-of-the-art approaches [15, 19, 20, 28] to address the respective problems.

2. Background and Motivation

In this section, we provide a brief background on DRAM organization and operation in order to understand the slowdown estimation mechanisms proposed by previous work. We then describe previous work on slowdown estimation and their key drawbacks that motivate our MISE model.

2.1. DRAM Organization

Modern DRAM main memory system is organized hierarchically into channels, ranks and banks. The main memory system consists of multiple channels that operate independently. Each channel consists of one or more ranks that share the channel's address and data buses. Each rank consists of multiple banks that share rank-level peripheral circuitry.

Banks can be viewed as arrays of DRAM cells, organized as rows and columns. To access a piece of data, the entire row containing the data is read into an internal buffer called the *row-buffer*. Subsequent accesses to the same row do not need to access the DRAM array, assuming the row is still in the row-buffer, and can be served faster.

2.2. Prior Work on Slowdown Estimation

While a large body of previous work has focused on main memory interference reduction techniques such as memory request scheduling [2, 19, 20, 29, 30], memory channel/bank partitioning [16, 27] and interleaving [18] to mitigate interference between applications at the DRAM channels, banks and row-buffers, few previous works have proposed techniques to estimate memory-interference-induced slowdowns.

Stall Time Fair Memory Scheduling (STFM) [28] is one previous work that attempts to estimate each application's slowdown, with the goal of improving fairness by prioritizing the most slowed down application. STFM estimates an application's slowdown as the ratio of its memory stall time when it is run alone versus when it is concurrently run alongside other applications. The challenge is in determining the alone-stall-time of an application *while* the application is actually running alongside other applications. STFM proposes to address this challenge by counting the number of cycles an application is stalled due to interference from other applications at the DRAM channels, banks and row-buffers. STFM uses this interference cycle count to estimate the alone-stall-time of the application, and hence the application's slowdown.

Fairness via Source Throttling (FST) [6] estimates application slowdowns due to inter-application interference at the shared caches and memory, as the ratio of uninterfered to interfered execution times. It uses the slowdown estimates to make informed source throttling decisions, to improve fairness. The mechanism to account for memory interference to estimate uninterfered execution time is similar to that employed in STFM.

A concurrent work by Du Bois et al. [5] proposes a mechanism to determine an application's standalone execution time when it shares cache and memory with other applications in a multicore system. In order to quantify memory interference, the mechanism proposed in this paper counts the number of waiting cycles due to inter-application interference and factors out these waiting cycles to estimate alone execution times, which is similar to STFM's alone stall time estimation mechanism.

Both of these works, FST [6] and Du Bois et al.'s mechanism [5], in principle, are similar to STFM from the perspective of quantifying the effect of memory interference on performance in that they estimate the *additional cycles* due to memory interference *while* an application is running alongside other applications. Since our focus in this work is estimating application slowdowns due to memory interference, we will focus on STFM in the next sections.

2.3. Motivation and Our Goal

In this work, we find that STFM's slowdown estimation model is inaccurate. The main reason for this is that quantifying the effect of interference, especially in a modern system employing out-of-order execution, is difficult. This is because even if a request is delayed due to interference, it may not affect performance due to memory-level parallelism, as the additional latency can be hidden by another outstanding request's latency. However, we note that the primary goal of STFM is not to estimate the slowdown of applications accurately, rather to use the estimates to make prioritization/throttling decisions to improve overall fairness. We provide a detailed qualitative and quantitative comparison between STFM and our model in Section 6.

Our goal, in this work, is to design a model to accurately estimate application slowdowns. Accurate slowdown estimates can enable various hardware/software techniques to enforce QoS guarantees. For instance, naive policies for QoS enforcement either execute an application entirely by itself or always prioritize a QoS-critical application of interest in shared resources [15]. However, accurate slowdown estimates can enable the memory controller to employ more intelligent and sophisticated memory bandwidth management policies. For instance, the memory controller could provide soft QoS guarantees by allocating just enough bandwidth to QoS-critical applications of interest, while utilizing the remaining memory bandwidth to achieve high overall system performance, as we will demonstrate in Section 8.1. Furthermore, higher accuracy slowdown estimates enable the memory controller to enforce system fairness more effectively than previous proposals, as we show in Section 8.2. Alternatively, information about application slowdowns could also be conveyed to the OS, enabling it to make better job scheduling decisions.

3. The MISE Model

In this section, we provide a detailed description of our proposed Memory-Interference-induced Slowdown Estimation (MISE) model. For ease of understanding, we first describe the observations that lead to a simple model for estimating the slowdown of a memory-bound application when it is run concurrently with other applications. In Section 3.2, we describe how we extend the model to accommodate non-memory-bound applications. Section 4 describes the detailed implementation of our model in a memory controller.

3.1. Memory-bound Application

A memory-bound application is one that spends an overwhelmingly large fraction of its execution time stalling on memory accesses. Therefore, the rate at which such an application's requests are served has significant impact on its performance. More specifically, we make the following observation about a memory-bound application.

> **Observation 1:** *The performance of a memory-bound application is roughly proportional to the rate at which its memory requests are served.*

For instance, for an application that is bottlenecked at memory, if the rate at which its requests are served is reduced by half, then the application will take twice as much time to finish the same amount of work. To validate this observation, we conducted a real-system experiment where we ran memory-bound applications from SPEC CPU2006 [1] on a 4-core Intel Core i7 [12]. Each SPEC application was run along with three copies of a microbenchmark whose memory intensity can be varied.[1] By varying the memory intensity of the microbenchmark, we can change the rate at which the requests of the SPEC application are served.

Figure 1 plots the results of this experiment for three memory-intensive SPEC benchmarks, namely, *mcf*, *omnetpp*, and *astar*. The figure shows the performance of each application vs. the rate at which its requests are served. The request service rate and performance are normalized to the request service rate and performance respectively of each application when it is run alone on the same system.

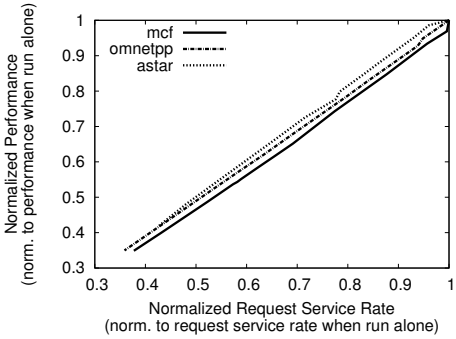


Figure 1: Request service rate vs. performance

The results of our experiments validate our observation. The performance of a memory-bound application is directly proportional to the rate at which its requests are served. This suggests that we can use the request-service-rate of an application as a proxy for its performance. More specifically, we can compute the slowdown of an application, i.e., the ratio of its performance when it is run alone on a system vs. its performance when it is run alongside other applications on the same system, as follows:

$$\text{Slowdown of an App.} = \frac{\textit{alone-request-service-rate}}{\textit{shared-request-service-rate}} \quad (1)$$

Estimating the *shared-request-service-rate* (SRSR) of an application is straightforward. It just requires the memory controller to keep track of how many requests of the application are served in

[1] The microbenchmark streams through a large region of memory (one block at a time). The memory intensity of the microbenchmark (LLC MPKI) is varied by changing the amount of computation performed between memory operations.

a given number of cycles. However, the challenge is to estimate the *alone-request-service-rate* (ARSR) of an application *while* it is run alongside other applications. A naive way of estimating ARSR of an application would be to prevent all other applications from accessing memory for a length of time and measure the application's ARSR. While this would provide an accurate estimate of the application's ARSR, this approach would significantly slow down other applications in the system. Our second observation helps us to address this problem.

Observation 2: *The ARSR of an application can be estimated by giving the requests of the application the highest priority in accessing memory.*

Giving an application's requests the highest priority in accessing memory results in very little interference from the requests of other applications. Therefore, many requests of the application are served as if the application were the only one running on the system. Based on the above observation, the ARSR of an application can be computed as follows:

$$\text{ARSR of an App.} = \frac{\text{\# Requests with Highest Priority}}{\text{\# Cycles with Highest Priority}} \quad (2)$$

where *# Requests with Highest Priority* is the number of requests served when the application is given highest priority, and *# Cycles with Highest Priority* is the number of cycles an application is given highest priority by the memory controller.

The memory controller can use Equation 2 to periodically estimate the ARSR of an application and Equation 1 to measure the slowdown of the application using the estimated ARSR. Section 4 provides a detailed description of the implementation of our model inside a memory controller.

3.2. Non-memory-bound Application

So far, we have described our MISE model for a memory-bound application. We find that the model presented above has low accuracy for non-memory-bound applications. This is because a non-memory-bound application spends significant fraction of its execution time in the *compute phase* (when the core is not stalled waiting for memory). Hence, varying the request service rate for such an application will not affect the length of the large compute phase. Therefore, we take into account the duration of the compute phase to make the model accurate for non-memory-bound applications.

Let α be the fraction of time spent by an application at memory. Therefore, the fraction of time spent by the application in the compute phase is $1 - \alpha$. Since changing the request service rate affects only the memory phase, we augment Equation 1 to take into account α as follows:

$$\textit{Slowdown of an App.} = (1 - \alpha) + \alpha \frac{\text{ARSR}}{\text{SRSR}} \quad (3)$$

In addition to estimating ARSR and SRSR required by Equation 1, the above equation requires estimating the parameter α, the fraction of time spent in memory phase. However, precisely computing α for a modern out-of-order processor is a challenge since such a processor overlaps computation with memory accesses. The processor stalls waiting for memory only when the oldest instruction in the reorder buffer is waiting on a memory request. For this reason, we estimate α as the fraction of time the processor spends stalling for memory.

$$\alpha = \frac{\text{\# Cycles spent stalling on memory requests}}{\text{Total number of cycles}} \quad (4)$$

Setting α to 1 reduces Equation 3 to Equation 1. We find that when even when an application is moderately memory-intensive, setting α to 1 provides a better estimate of slowdown. Therefore, our final model for estimating slowdown takes into account the stall fraction (α) only when it is low. Algorithm 1 shows our final slowdown estimation model.

Compute α;
if $\alpha < \textit{Threshold}$ **then**
    Slowdown $= (1 - \alpha) + \alpha \frac{\text{ARSR}}{\text{SRSR}}$
else
    Slowdown $= \frac{\text{ARSR}}{\text{SRSR}}$
end

Algorithm 1: The MISE model

4. Implementation

In this section, we describe a detailed implementation of our MISE model in a memory controller. For each application in the system, our model requires the memory controller to compute three parameters: 1) *shared-request-service-rate* (SRSR), 2) *alone-request-service-rate* (ARSR), and 3) α (stall fraction).[2] First, we describe the scheduling algorithm employed by the memory controller. Then, we describe how the memory controller computes each of the three parameters.

4.1. Memory Scheduling Algorithm

In order to implement our model, each application needs to be given the highest priority periodically, such that its *alone-request-service-rate* can be measured. This can be achieved by simply assigning each application's requests highest priority in a round-robin manner. However, the mechanisms we build on top of our model allocate bandwidth to different applications to achieve QoS/fairness. Therefore, in order to facilitate the implementation of our mechanisms, we employ a lottery-scheduling-like approach [32, 36] to schedule requests in the memory controller. The basic idea of lottery scheduling is to probabilistically enforce a given bandwidth allocation, where each application is allocated a certain share of the system bandwidth. The exact bandwidth allocation policy depends on the goal of the system – e.g., QoS, high performance, high fairness, etc. In this section, we describe how a lottery-scheduling-like algorithm works to enforce a bandwidth allocation.

The memory controller divides execution time into *intervals* (of $\mathcal{M}$ processor cycles each). Each interval is further divided into small *epochs* (of $\mathcal{N}$ processor cycles each). At the beginning of each *interval*, the memory controller estimates the slowdown of each application in the system. Based on the slowdown estimates and the final goal, the controller may change the bandwidth allocation policy – i.e., redistribute bandwidth amongst the concurrently running applications. At the beginning of each *epoch*, the memory controller probabilistically picks a single application and prioritizes all the requests of that particular application during that epoch. The probability distribution used to choose the prioritized application is such that an application with higher bandwidth allocation has a higher probability of getting the highest priority. For example, consider a system with two applications, A and B. If the memory controller allocates A 75% of the memory bandwidth and B the remaining 25%, then A and B get the highest priority with probability 0.75 and 0.25, respectively.

[2] These three parameters need to be computed only for the active applications in the system. Hence, these need to be tracked only per hardware thread context.

4.2. Computing *shared-request-service-rate* (SRSR)

The *shared-request-service-rate* of an application is the rate at which the application's requests are served while it is running with other applications. This can be directly computed by the memory controller using a per-application counter that keeps track of the number of requests served for that application. At the beginning of each *interval*, the controller resets the counter for each application. Whenever a request of an application is served, the controller increments the counter corresponding to that application. At the end of each *interval*, the SRSR of an application is computed as

$$\text{SRSR of an App} = \frac{\#\text{ Requests served}}{\mathcal{M}\text{ (Interval Length)}}$$

4.3. Computing *alone-request-service-rate* (ARSR)

The *alone-request-service-rate* (ARSR) of an application is an estimate of the rate at which the application's requests would have been served had it been running alone on the same system. Based on our observation (described in Section 3.1), the ARSR can be estimated by using the request-service-rate of the application when its requests have the highest priority in accessing memory. Therefore, the memory controller estimates the ARSR of an application only during the *epochs* in which the application has the highest priority.

Ideally, the memory controller should be able to achieve this using two counters: one to keep track of the number of *epochs* during which the application received highest priority and another to keep track of the number of requests of the application served during its highest-priority *epochs*. However, it is possible that even when an application's requests are given highest priority, they may receive interference from other applications' requests. This is because, our memory scheduling is *work conserving* – if there are no requests from the highest priority application, it schedules a ready request from some other application. Once a request is scheduled, it cannot be preempted because of the way DRAM operates.

In order to account for this interference, the memory controller uses a third counter for each application to track the number of cycles during which an application's request was blocked due to some other application's request, in spite of the former having highest priority. For an application with highest priority, a cycle is deemed to be an *interference cycle* if during that cycle, a command corresponding to a request of that application is waiting in the request buffer and the previous command issued to any bank, was for a request from a different application.

Based on the above discussion, the memory controller keeps track of three counters to compute the ARSR of an application: 1) number of highest-priority *epochs* of the application (# HPEs), 2) number of requests of that application served during its highest-priority *epochs* (# HPE Requests), and 3) number of *interference cycles* of the application during its highest-priority *epochs* (# Interference cycles). All these counters are reset at the start of an *interval* and the ARSR is computed at the end of each interval as follows:

$$\text{ARSR of an App.} = \frac{\#\text{ HPE Requests}}{\mathcal{N}.(\#\text{ HPEs}) - (\#\text{ Interference cycles})}$$

Our model does not take into account bank level parallelism (BLP) or row-buffer interference when estimating # Interference cycles. We observe that this does not affect the accuracy of our model significantly. because we eliminate most of the interference by measuring ARSR only when an application has highest priority. We leave a study of the effects of bank-level parallelism and row-buffer interference on the accuracy of our model as part of future work.

4.4. Computing *stall-fraction* α

The *stall-fraction* (α) is the fraction of the cycles spent by the application stalling for memory requests. The number of stall cycles can be easily computed by the core and communicated to the memory controller at the end of each interval.

4.5. Hardware Cost

Our implementation incurs additional storage cost due to 1) the counters that keep track of parameters required to compute slowdown (five per hardware thread context), and 2) a register that keeps track of the current bandwidth allocation policy (one per hardware thread context). We find that using four byte registers for each counter is more than sufficient for the values they keep track of. Therefore, our model incurs a storage cost of at most 24 bytes per hardware thread context.

5. Methodology

Simulation Setup. We model the memory system using an in-house cycle-accurate DDR3-SDRAM simulator. We have integrated this DDR3 simulator into an in-house cycle-level x86 simulator with a Pin [21] frontend, which models out-of-order cores with a limited-size instruction window. Each core has a 512 KB private cache. We model main memory as the only shared resource, in order to isolate and analyze the effect of memory interference on application slowdowns. Table 1 provides more details of the simulated systems.

Unless otherwise specified, the evaluated systems consist of 4 cores and a memory subsystem with 1 channel, 1 rank/channel and 8 banks/rank. We use row-interleaving to map the physical address space onto DRAM channels, ranks and banks. Data is striped across different channels, ranks and banks, at the granularity of a row. Our workloads are made up of 26 benchmarks from the SPEC CPU2006 [1] suite.

Workloads. We form multiprogrammed workloads using combinations of these 26 benchmarks. We extract a representative phase of each benchmark using PinPoints [31] and run that phase for 200 million cycles. We will provide more details about our workloads as and when required.

Processor	4-16 cores, 5.3GHz, 3-wide issue, 8 MSHRs, 128-entry instruction window
Last-level cache	64B cache-line, 16-way associative, 512kB private cache-slice per core
Memory controller	64/64-entry read/write request queues per controller
Memory	Timing: DDR3-1066 (8-8-8) [25] Organization: 1 channel, 1 rank-per-channel, 8 banks-per-rank, 8 KB row-buffer

Table 1: Configuration of the simulated system

Metrics. We compute both weighted speedup [34] and harmonic speedup [22] to measure system performance. However, since the goal of our mechanisms is to provide QoS/high fairness, while ensuring good system performance, we mainly present harmonic speedup throughout our evaluations, as the harmonic speedup metric provides a good balance between system performance and fairness [22]. We use the maximum slowdown [4, 19, 20] metric to measure unfairness.

Parameters. We use an interval length ($\mathcal{M}$) of 5 million cycles and an epoch length ($\mathcal{N}$) of 10000 cycles for all our evaluations. Section 7 evaluates sensitivity of our model to these parameters.

6. Comparison to STFM

Stall-Time-Fair Memory scheduling (STFM) [28] is one of the few previous works that attempt to estimate main-memory-induced slowdowns of individual applications when they are run concurrently on a multicore system. As we described in Section 2, STFM estimates the slowdown of an application by estimating the number of cycles it stalls due to interference from other applications' requests. In this section, we qualitatively and quantitatively compare MISE with STFM.

There are two key differences between MISE and STFM for estimating slowdown. First, MISE uses request service rates rather than stall times to estimate slowdown. As we mentioned in Section 3, the *alone-request-service-rate* of an application can be fairly accurately estimated by giving the application highest priority in accessing memory. Giving the application highest priority in accessing memory results in very little interference from other applications. In contrast, STFM attempts to estimate the alone-stall-time of an application while it is receiving significant interference from other applications. Second, MISE takes into account the effect of the compute phase for non-memory-bound applications. STFM, on the other hand, has no such provision to account for the compute phase. As a result, MISE's slowdown estimates for non-memory-bound applications are significantly more accurate than STFM's estimates.

Figure 2 compares the accuracy of the MISE model with STFM for six representative memory-bound applications from the SPEC CPU2006 benchmark suite. Each application is run on a 4-core system along with three other applications: *sphinx3*, *leslie3d*, and *milc*. The figure plots three curves: 1) actual slowdown, 2) slowdown estimated by STFM, and 3) slowdown estimated by MISE. For most applications in the SPEC CPU2006 suite, the slowdown estimated by MISE is significantly more accurate than STFM's slowdown estimates. All applications whose slowdowns are shown in Figure 2, except sphinx3 are representative of this behavior. For a few applications and workload combinations, STFM's estimates are comparable to the slowdown estimates from our model. *sphinx3* is an example of such an application. However, as we will show below, across all workloads, the MISE model provides lower average slowdown estimation error for all applications.

Figure 3 compares the accuracy of MISE with STFM for three representative non-memory-bound applications, when each application is run on a 4-core system along with three other applications: *sphinx3*, *leslie3d*, and *milc*. As shown in the figure, MISE's estimates are significantly more accurate compared to STFM's estimates. As mentioned before, STFM does not account for the compute phase of these applications. However, these applications spend significant amount of their execution time in the compute phase. This is the reason why our model, which takes into account the effect of the compute phase of these applications, is able to provide more accurate slowdown estimates for non-memory-bound applications.

Table 2 shows the average slowdown estimation error for each benchmark, with STFM and MISE, across all 300 4-core workloads of different memory intensities.[3] As can be observed, MISE's slowdown estimates have significantly lower error than STFM's slowdown estimates across most benchmarks. Across *300* workloads, STFM's estimates deviate from the actual slowdown by 29.8%, whereas, our proposed MISE model's estimates deviate from the actual slowdown by only 8.1%. Therefore, we conclude that our slowdown estimation model provides better accuracy than STFM.

[3] See Table 4 and Section 8.1.3 for more details about these 300 workloads.

Benchmark	STFM	MISE	Benchmark	STFM	MISE
453.povray	56.3	0.1	473.astar	12.3	8.1
454.calculix	43.5	1.3	456.hmmer	17.9	8.1
400.perlbench	26.8	1.6	464.h264ref	13.7	8.3
447.dealII	37.5	2.4	401.bzip2	28.3	8.5
436.cactusADM	18.4	2.6	458.sjeng	21.3	8.8
450.soplex	29.8	3.5	433.milc	26.4	9.5
444.namd	43.6	3.7	481.wrf	33.6	11.1
437.leslie3d	26.4	4.3	429.mcf	83.74	11.5
403.gcc	25.4	4.5	445.gobmk	23.1	12.5
462.libquantum	48.9	5.3	483.xalancbmk	18.0	13.6
459.GemsFDTD	21.6	5.5	435.gromacs	31.4	15.6
470.lbm	6.9	6.3	482.sphinx3	21	16.8
473.astar	12.3	8.1	471.omnetpp	26.2	17.5
456.hmmer	17.9	8.1	465.tonto	32.7	19.5

Table 2: Average error for each benchmark (in %)

7. Sensitivity to Algorithm Parameters

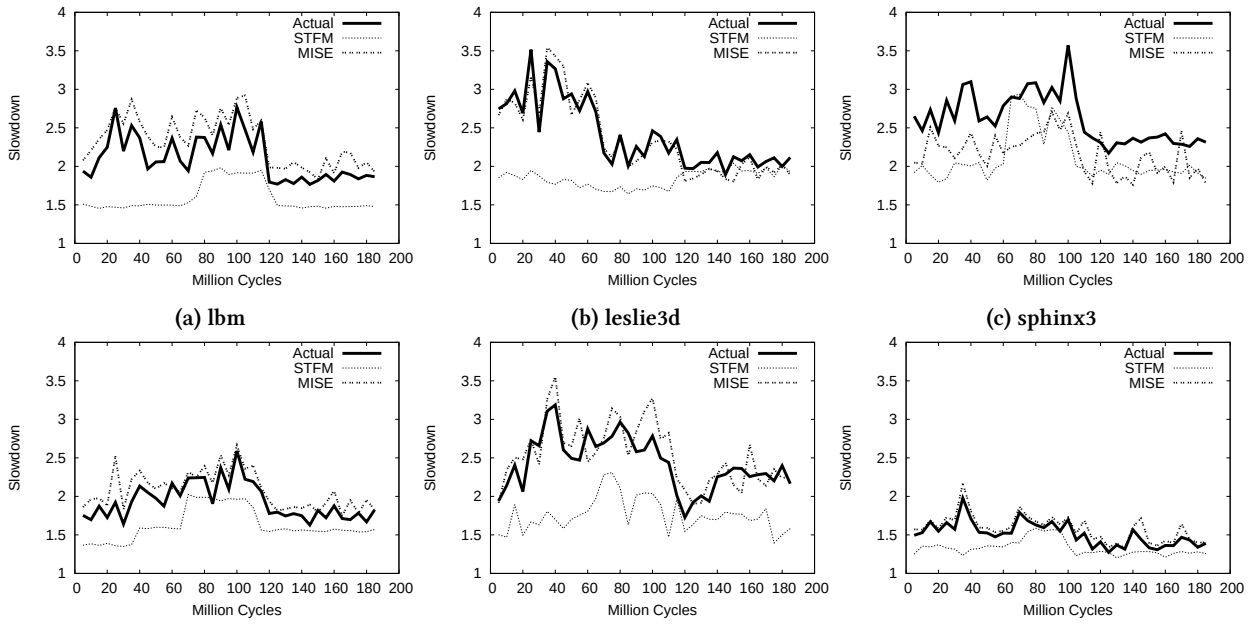
We evaluate the sensitivity of the MISE model to epoch and interval lengths. Table 3 presents the average error (in %) of the MISE model for different values of epoch and interval lengths. Two major conclusions are in order. First, when the interval length is small (1 million cycles), the error is very high. This is because the request service rate is not stable at such small interval lengths and varies significantly across intervals. Therefore, it cannot serve as an effective proxy for performance. On the other hand, when the interval length is larger, request service rate exhibits a more stable behavior and can serve as an effective measure of application slowdowns. Therefore, we conclude that except at very low interval lengths, the MISE model is robust. Second, the average error is high for high epoch lengths (1 million cycles) because the number of epochs in an interval reduces. As a result, some applications might not be assigned highest priority for any epoch during an interval, preventing estimation of their *alone-request-service-rate*. Note that the effect of this is mitigated as the interval length increases, as with a larger interval length the number of epochs in an interval increases. For smaller epoch length values, however, the average error of MISE does not exhibit much variation and is robust. The lowest average error of 8.1% is achieved at an interval length of 5 million cycles and an epoch length of 10000 cycles. Furthermore, we observe that estimating slowdowns at an interval length of 5 million cycles also enables enforcing QoS at fine time granularities, although, higher interval lengths exhibit similar average error. Therefore, we use these values of interval and epoch lengths for our evaluations.

Epoch Length \ Interval Length	1 mil.	5 mil.	10 mil.	25 mil.	50 mil.
1000	65.1%	9.1%	11.5%	10.7%	8.2%
10000	64.1%	8.1%	9.6%	8.6%	8.5%
100000	64.3%	11.2%	9.1%	8.9%	9%
1000000	64.5%	31.3%	14.8%	14.9%	11.7%

Table 3: Sensitivity to epoch and interval lengths

8. Leveraging the MISE Model

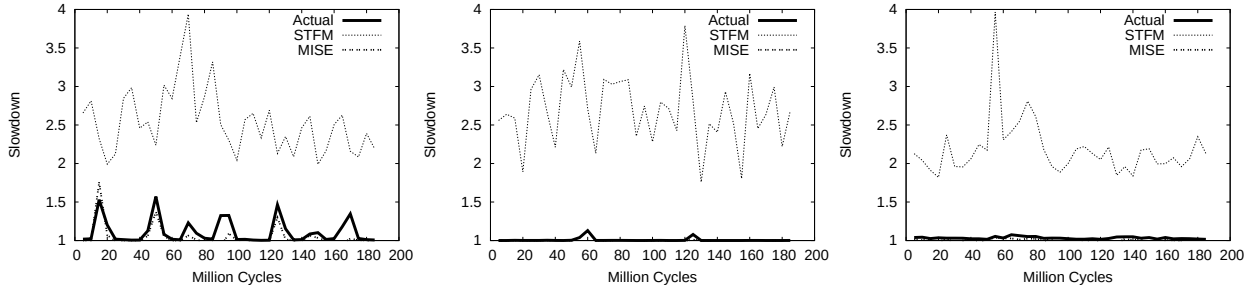
Our MISE model for estimating individual application slowdowns can be used to design different policies to better en-



(a) lbm (b) leslie3d (c) sphinx3

(d) GemsFDTD (e) soplex (f) cactusADM

Figure 2: Comparison of our MISE model with STFM for representative memory-bound applications



(a) wrf (b) povray (c) calculix

Figure 3: Comparison of our MISE model with STFM for representative non-memory-bound applications

force quality-of-service (QoS) and fairness. In this section, we describe and evaluate two such mechanisms: 1) a mechanism to provide soft QoS guarantees (MISE-QoS) and 2) a mechanism that attempts to minimize maximum slowdown to improve overall system fairness (MISE-Fair).

8.1. MISE-QoS: Providing Soft QoS Guarantees

MISE-QoS is a mechanism to provide soft QoS guarantees to one or more applications of interest in a workload with many applications, while trying to maximize overall performance for the remaining applications. By *soft QoS guarantee*, we mean that the applications of interest (AoIs) should not be slowed down by more than an operating-system-specified bound. A naive way of achieving such a soft QoS guarantee is to always prioritize the AoIs. However, such a mechanism has two shortcomings. First, the naive mechanism can work when there is only one AoI. With more than one AoI, prioritizing all AoIs will cause them to interfere with each other making their slowdowns uncontrollable. Second, even with just one AoI, the naive mechanism may unnecessarily slow down other applications in the system by excessively prioritizing the AoI. MISE-QoS addresses these shortcomings by using slowdown estimates of the AoIs to allocate them just enough memory bandwidth to meet their specified slowdown bound. We present the operation of MISE-QoS with one AoI and then describe how it can be extended to multiple AoIs.

8.1.1. Mechanism Description. The operation of MISE-QoS with one AoI is simple. As described in Section 4.1, the memory controller divides execution time into intervals of length $\mathcal{M}$. The controller maintains the current bandwidth allocation for the AoI. At the end of each interval, it estimates the slowdown of the AoI and compares it with the specified bound, say B. If the estimated slowdown is less than B, then the controller reduces the bandwidth allocation for the AoI by a small amount (2% in our experiments). On the other hand, if the estimated slowdown is more than B, the controller increases the bandwidth allocation for the AoI (by 2%).[4] The remaining bandwidth is used by all other applications in the system in a free-for-all manner. The above mechanism attempts to ensure that the AoI gets just enough bandwidth to meet its target slowdown bound. As a result, the other applications in the system are not *unnecessarily* slowed down.

In some cases, it is possible that the target bound *cannot* be met even by allocating all the memory bandwidth to the AoI – i.e., prioritizing its requests 100% of the time. This is because, even the application with the highest priority (AoI) could be subject to interference, slowing it down by some factor, as we describe in Section 4.3. Therefore, in scenarios when it is

[4] We found that 2% increments in memory bandwidth work well empirically, as our results indicate. Better techniques that dynamically adapt the increment are possible and are a part of our future work.

Mix No.	Benchmark 1	Benchmark 2	Benchmark 3
1	sphinx3	leslie3d	milc
2	sjeng	gcc	perlbench
3	tonto	povray	wrf
4	perlbench	gcc	povray
5	gcc	povray	leslie3d
6	perlbench	namd	lbm
7	hef	bzip2	libquantum
8	hmmer	lbm	omnetpp
9	sjeng	libquantum	cactusADM
10	namd	libquantum	mcf
11	xalancbmk	mcf	astar
12	mcf	libquantum	leslie3d

Table 4: Workload mixes

not possible to meet the target bound for the AoI, the memory controller can convey this information to the operating system, which can then take appropriate action (e.g., deschedule some other applications from the machine).

8.1.2. MISE-QoS with Multiple AoIs. The above described MISE-QoS mechanism can be easily extended to a system with multiple AoIs. In such a system, the memory controller maintains the bandwidth allocation for each AoI. At the end of each interval, the controller checks if the slowdown estimate for each AoI meets the corresponding target bound. Based on the result, the controller either increases or decreases the bandwidth allocation for each AoI (similar to the mechanism in Section 8.1.1).

With multiple AoIs, it may not be possible to meet the specified slowdown bound for any of the AoIs. Our mechanism concludes that the specified slowdown bounds cannot be met if: 1) all the available bandwidth is partitioned only between the AoIs – i.e., no bandwidth is allocated to the other applications, and 2) any of the AoIs does not meet its slowdown bound after R intervals (where R is empirically determined at design time). Similar to the scenario with one AoI, the memory controller can convey this conclusion to the operating system (along with the estimated slowdowns), which can then take an appropriate action. Note that other potential mechanisms for determining whether slowdown bounds can be met are possible.

8.1.3. Evaluation with Single AoI. To evaluate MISE-QoS with a single AoI, we run each benchmark as the AoI, alongside 12 different workload mixes shown in Table 4. We run each workload with 10 different slowdown bounds for the AoI: $\frac{10}{1}, \frac{10}{2}, ..., \frac{10}{10}$. These slowdown bounds are chosen so as to have more data points between the bounds of $1\times$ and $5\times$.[5] In all, we present results for 3000 data points with different workloads and slowdown bounds. We compare MISE-QoS with a mechanism that always prioritizes the AoI [15].

Table 5 shows the effectiveness of MISE-QoS in meeting the prescribed slowdown bounds for the 3000 data points. As shown, for approximately 79% of the workloads, MISE-QoS meets the specified bound and correctly predicts that the bound is met. However, for 2.1% of the workloads, MISE-QoS *does* meet the specified bound but it incorrectly predicts that the bound is not met. This is because, in some cases, MISE-QoS slightly overestimates the slowdown of applications. Overall, MISE-QoS meets the specified slowdown bound for close to 80.9% of the workloads, as compared to *AlwaysPrioritize* that meets the bound for 83% of the workloads. Therefore, we conclude that MISE-QoS meets the bound for 97.5% of the workloads where *AlwaysPrioritize* meets the bound. Furthermore, MISE-QoS correctly predicts whether or not the bound was met for 95.7% of the workloads, whereas *AlwaysPrioritize* has no provision to predict whether or not the bound was met.

[5]Most applications are not slowed down by more than $5\times$ for our system configuration.

Scenario	# Workloads	% Workloads
Bound Met and Predicted Right	2364	78.8%
Bound Met and Predicted Wrong	65	2.1%
Bound Not Met and Predicted Right	509	16.9%
Bound Not Met and Predicted Wrong	62	2.2%

Table 5: Effectiveness of MISE-QoS

To show the effectiveness of MISE-QoS, we compare the AoI's slowdown due to MISE-QoS and the mechanism that always prioritizes the AoI (*AlwaysPrioritize*) [15]. Figure 4 presents representative results for 8 different AoIs when they are run alongside Mix 1 (Table 4). The label MISE-QoS-n corresponds to a slowdown bound of $\frac{10}{n}$. (Note that *AlwaysPrioritize* does not take into account the slowdown bound). Note that the slowdown bound decreases (i.e., becomes tighter) from left to right for each benchmark in Figure 4 (as well as in other figures). We draw three conclusions from the results.

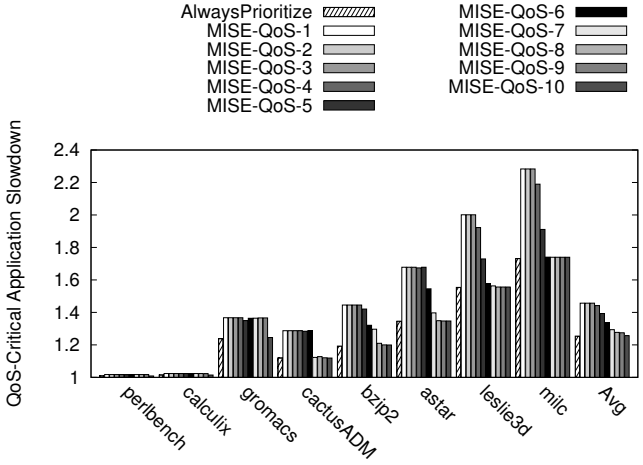


Figure 4: AoI performance: MISE-QoS vs. *AlwaysPrioritize*

First, for most applications, the slowdown of *AlwaysPrioritize* is considerably more than one. As described in Section 8.1.1, always prioritizing the AoI does not completely prevent other applications from interfering with the AoI.

Second, as the slowdown bound for the AoI is decreased (left to right), MISE-QoS gradually increases the bandwidth allocation for the AoI, eventually allocating all the available bandwidth to the AoI. At this point, MISE-QoS performs very similarly to the *AlwaysPrioritize* mechanism.

Third, in almost all cases (in this figure and across all our 3000 data points), MISE-QoS meets the specified slowdown bound *if AlwaysPrioritize* is able to meet the bound. One exception to this is benchmark *gromacs*. For this benchmark, MISE-QoS meets the slowdown bound for values ranging from $\frac{10}{1}$ to $\frac{10}{6}$.[6] For slowdown bound values of $\frac{10}{7}$ and $\frac{10}{8}$, MISE-QoS does not meet the bound even though allocating all the bandwidth for *gromacs* would have achieved these slowdown bounds (since *AlwaysPrioritize* can meet the slowdown bound for these values). This is because our MISE model underestimates the slowdown for *gromacs*. Therefore, MISE-QoS incorrectly assumes that the slowdown bound is met for *gromacs*.

Overall, MISE-QoS accurately estimates the slowdown of the AoI and allocates just enough bandwidth to the AoI to meet a

[6]Note that the slowdown bound becomes tighter from left to right.

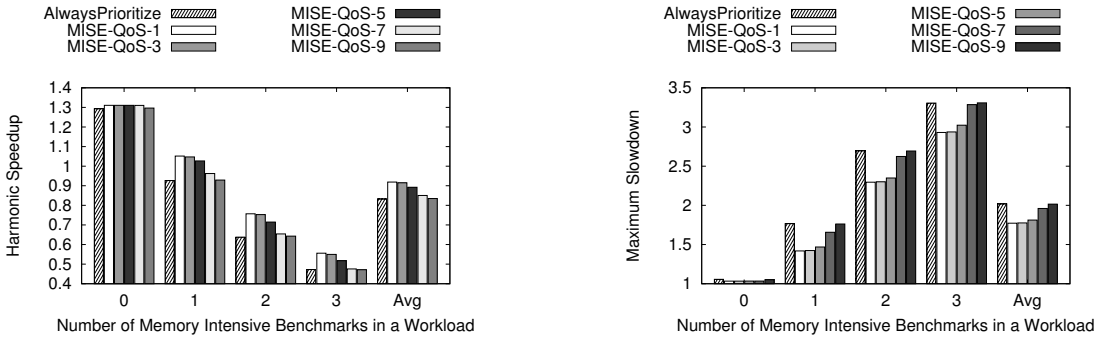


Figure 5: Average system performance and fairness across 300 workloads of different memory intensities

slowdown bound. As a result, MISE-QoS is able to significantly improve the performance of the other applications in the system (as we show next).

System Performance and Fairness. Figure 5 compares the system performance (harmonic speedup) and fairness (maximum slowdown) of MISE-QoS and *AlwaysPrioritize* for different values of the bound. We omit the AoI from the performance and fairness calculations. The results are categorized into four workload categories (0, 1, 2, 3) indicating the number of memory-intensive benchmarks in the workload. For clarity, the figure shows results only for a few slowdown bounds. Three conclusions are in order.

First, MISE-QoS significantly improves performance compared to *AlwaysPrioritize*, especially when the slowdown bound for the AoI is large. On average, when the bound is $\frac{10}{3}$, MISE-QoS improves harmonic speedup by 12% and weighted speedup by 10% (not shown due to lack of space) over *AlwaysPrioritize*, while reducing maximum slowdown by 13%. Second, as expected, the performance and fairness of MISE-QoS approach that of *AlwaysPrioritize* as the slowdown bound is decreased (going from left to right for a set of bars). Finally, the benefits of MISE-QoS increase with increasing memory intensity because always prioritizing a memory intensive application will cause significant interference to other applications.

Based on our results, we conclude that MISE-QoS can effectively ensure that the AoI meets the specified slowdown bound while achieving high system performance and fairness across the other applications. In Section 8.1.4, we discuss a case study of a system with two AoIs.

Using STFM's Slowdown Estimates to Provide QoS. We evaluate the effectiveness of STFM in providing slowdown guarantees, by using slowdown estimates from STFM's model to drive our QoS-enforcement mechanism. Table 6 shows the effectiveness of STFM's slowdown estimation model in meeting the prescribed slowdown bounds for the 3000 data points. We draw two major conclusions. First, the slowdown bound is met and predicted right for only 63.7% of the workloads, whereas MISE-QoS meets the slowdown bound and predicts it right for 78.8% of the workloads (as shown in Table 5). The reason is STFM's high slowdown estimation error. Second, the percentage of workloads for which the slowdown bound is met/not-met and is predicted wrong is 18.4%, as compared to 4.3% for MISE-QoS. This is because STFM's slowdown estimation model overestimates the slowdown of the AoI and allocates it more bandwidth than is required to meet the prescribed slowdown bound. Therefore, performance of the other applications in a workload suffers, as demonstrated in Figure 6 which shows the system performance for different values of the prescribed slowdown bound, for MISE and STFM. For instance, when the slowdown bound is $\frac{10}{3}$, STFM-QoS has 5% lower average system performance than MISE-QoS. Therefore, we conclude that the proposed MISE model enables more effective enforcement of QoS guarantees for the AoI, than the STFM model, while providing better average system performance.

Scenario	# Workloads	% Workloads
Bound Met and Predicted Right	1911	63.7%
Bound Met and Predicted Wrong	480	16%
Bound Not Met and Predicted Right	537	17.9%
Bound Not Met and Predicted Wrong	72	2.4%

Table 6: Effectiveness of STFM-QoS

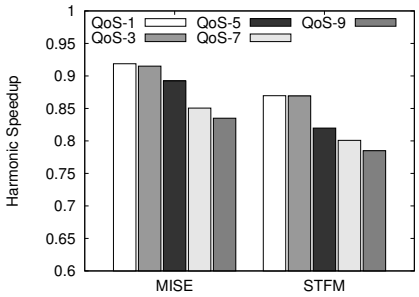


Figure 6: Average system performance using MISE and STFM's slowdown estimation models (across 300 workloads)

8.1.4. Case Study: Two AoIs. So far, we have discussed and analyzed the benefits of MISE-QoS for a system with one AoI. However, there could be scenarios with multiple AoIs each with its own target slowdown bound. One can think of two naive approaches to possibly address this problem. In the first approach, the memory controller can prioritize the requests of all AoIs in the system. This is similar to the *AlwaysPrioritize* mechanism described in the previous section. In the second approach, the memory controller can equally partition the memory bandwidth across all AoIs. We call this approach *EqualBandwidth*. However, neither of these mechanisms can guarantee that the AoIs meet their target bounds. On the other hand, using the mechanism described in Section 8.1.2, MISE-QoS can be used to achieve the slowdown bounds for multiple AoIs.

To show the effectiveness of MISE-QoS with multiple AoIs, we present a case study with two AoIs. The two AoIs, *astar* and *mcf* are run in a 4-core system with *leslie* and another copy of *mcf*. Figure 7 compares the slowdowns of each of the four applications with the different mechanisms. The same slowdown bound is used for both AoIs.

Although *AlwaysPrioritize* prioritizes both AoIs, *mcf* (the more memory-intensive AoI) interferes significantly with *astar* (slowing it down by more than 7×). *EqualBandwidth* mitigates this interference problem by partitioning the bandwidth between the two applications. However, MISE-QoS intelligently

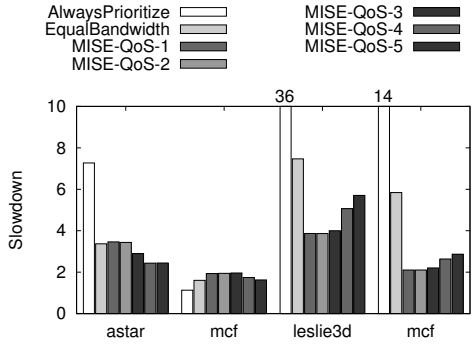


Figure 7: Meeting a target bound for two applications

partitions the available memory bandwidth equally between the two applications to ensure that both of them meet a more stringent target bound. For example, for a slowdown bound of $\frac{10}{4}$, MISE-QoS allocates more than 50% of the bandwidth to *astar*, thereby reducing *astar*'s slowdown below the bound of 2.5, while *EqualBandwidth* can only achieve a slowdown of 3.4 for astar, by equally partitioning the bandwidth between the two AoIs. Furthermore, as a result of its intelligent bandwidth allocation, MISE-QoS significantly reduces the slowdowns of the other applications in the system compared to *AlwaysPrioritize* and *EqualBandwidth* (as seen in Figure 7).

We conclude, based on the evaluations presented above, that MISE-QoS manages memory bandwidth efficiently to achieve both high system performance and fairness while meeting performance guarantees for one or more applications of interest.

8.2. MISE-Fair: Minimizing Maximum Slowdown

The second mechanism we build on top of our MISE model is one that seeks to improve overall system fairness. Specifically, this mechanism attempts to minimize the maximum slowdown across all applications in the system. Ensuring that no application is unfairly slowed down while maintaining high system performance is an important goal in multicore systems where co-executing applications are similarly important.

8.2.1. Mechanism. At a high level, our mechanism works as follows. The memory controller maintains two pieces of information: 1) a target slowdown bound (B) for all applications, and 2) a bandwidth allocation policy that partitions the available memory bandwidth across all applications. The memory controller enforces the bandwidth allocation policy using the lottery-scheduling technique as described in Section 4.1. The controller attempts to ensure that the slowdown of all applications is within the bound B. To this end, it modifies the bandwidth allocation policy so that applications that are slowed down more get more memory bandwidth. Should the memory controller find that bound B is not possible to meet, it increases the bound. On the other hand, if the bound is easily met, it decreases the bound. We describe the two components of this mechanism: 1) bandwidth redistribution policy, and 2) modifying target bound (B).

Bandwidth Redistribution Policy. As described in Section 4.1, the memory controller divides execution into multiple *intervals*. At the end of each interval, the controller estimates the slowdown of each application and possibly redistributes the available memory bandwidth amongst the applications, with the goal of minimizing the maximum slowdown. Specifically, the controller divides the set of applications into two clusters. The first cluster contains those applications whose estimated slowdown is less than B. The second cluster contains those applications whose estimated slowdown is more than B. The memory controller steals a small fixed amount of bandwidth allocation (2%) from each application in the first cluster and distributes it equally among the applications in the second cluster. This ensures that the applications that do not meet the target bound B get a larger share of the memory bandwidth.

Modifying Target Bound. The target bound B may depend on the workload and the different phases within each workload. This is because different workloads, or phases within a workload, have varying demands from the memory system. As a result, a target bound that is easily met for one workload/phase may not be achievable for another workload/phase. Therefore, our mechanism dynamically varies the target bound B by predicting whether or not the current value of B is achievable. For this purpose, the memory controller keeps track of the number of applications that met the slowdown bound during the past N intervals (3 in our evaluations). If all the applications met the slowdown bound in all of the N intervals, the memory controller predicts that the bound is easily achievable. In this case, it sets the new bound to a slightly lower value than the estimated slowdown of the application that is the most slowed down (a more competitive target). On the other hand, if more than half the applications did not meet the slowdown bound in all of the N intervals, the controller predicts that the target bound is not achievable. It then increases the target slowdown bound to a slightly higher value than the estimated slowdown of the most slowed down application (a more achievable target).

8.2.2. Interaction with the OS. As we will show in Section 8.2.3, our mechanism provides the best fairness compared to three state-of-the-art approaches for memory request scheduling [19, 20, 28]. In addition to this, there is another benefit to using our approach. Our mechanism, based on the MISE model, can accurately estimate the slowdown of each application. Therefore, the memory controller can potentially communicate the estimated slowdown information to the operating system (OS). The OS can use this information to make more informed scheduling and mapping decisions so as to further improve system performance or fairness. Since prior memory scheduling approaches do not explicitly attempt to minimize maximum slowdown by accurately estimating the slowdown of individual applications, such a mechanism to interact with the OS is not possible with them. Evaluating the benefits of the interaction between our mechanism and the OS is beyond the scope of this paper.

8.2.3. Evaluation. Figure 8 compares the system fairness (maximum slowdown) of different mechanisms with increasing number of cores. The figure shows results with four previously proposed memory scheduling policies (FRFCFS [33, 37], ATLAS [19], TCM [20], and STFM [28]), and our proposed mechanism using the MISE model (MISE-Fair). We draw three conclusions from our results.

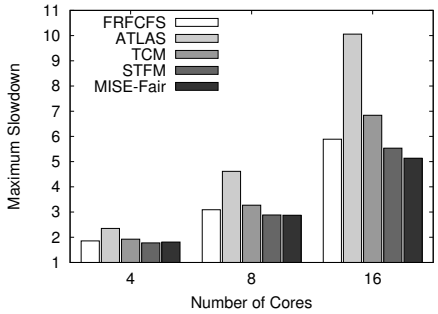


Figure 8: Fairness with different core counts

First, MISE-Fair provides the best fairness compared to all other previous approaches. The reduction in the maximum

slowdown due to MISE-Fair when compared to STFM (the best previous mechanism) increases with increasing number of cores. With 16 cores, MISE-Fair provides 7.2% better fairness compared to STFM.

Second, STFM, as a result of prioritizing the most slowed down application, provides better fairness than all other previous approaches. While the slowdown estimates of STFM are not as accurate as those of our mechanism, they are good enough to identify the most slowed down application. However, as the number of concurrently-running applications increases, simply prioritizing the most slowed down application may not lead to better fairness. MISE-Fair, on the other hand, works towards reducing maximum slowdown by stealing bandwidth from those applications that are less slowed down compared to others. As a result, the fairness benefits of MISE-Fair compared to STFM increase with increasing number of cores.

Third, ATLAS and TCM are more unfair compared to FRFCFS. As shown in prior work [19, 20], ATLAS trades off fairness to obtain better performance. TCM, on the other hand, is designed to provide high system performance and fairness. Further analysis showed us that the cause of TCM's unfairness is the strict ranking employed by TCM. TCM ranks all applications based on its clustering and shuffling techniques [20] and strictly enforces these rankings. We found that such strict ranking destroys the row-buffer locality of low-ranked applications. This increases the slowdown of such applications, leading to high maximum slowdown.

Effect of Workload Memory Intensity on Fairness. Figure 9 shows the maximum slowdown of the 16-core workloads categorized by workload intensity. While most trends are similar to those in Figure 8, we draw the reader's attention to a specific point: for workloads with non-memory-intensive applications (25%, 50% and 75% in the figure), STFM is more unfair than MISE-Fair. As shown in Figure 3, STFM significantly overestimates the slowdown of non-memory-bound applications. Therefore, for these workloads, we find that STFM prioritizes such non-memory-bound applications which are not the most slowed down. On the other hand, MISE-Fair, with its more accurate slowdown estimates, is able to provide better fairness for these workload categories.

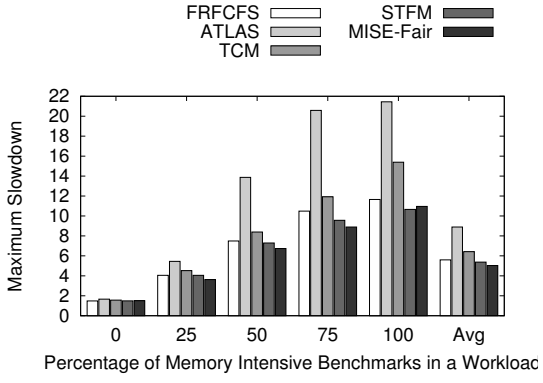


Figure 9: Fairness for 16-core workloads

System Performance. Figure 10 presents the harmonic speedup of the four previously proposed mechanisms (FRFCFS, ATLAS, TCM, STFM) and MISE-Fair, as the number of cores is varied. The results indicate that STFM provides the best harmonic speedup for 4-core and 8-core systems. STFM achieves this by prioritizing the most slowed down application. However, as the number of cores increases, the harmonic speedup of MISE-Fair matches that of STFM. This is because, with increasing number of cores, simply prioritizing the most slowed down application can be unfair to other applications. In contrast, MISE-Fair takes into account slowdowns of all applications to manage memory bandwidth in a manner that enables good progress for all applications. We conclude that MISE-Fair achieves the best fairness compared to prior approaches, without significantly degrading system performance.

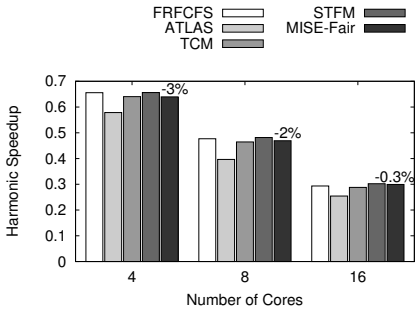


Figure 10: Harmonic speedup with different core counts

9. Related Work

To our knowledge, this is the first paper to 1) provide a *simple and accurate* model to estimate application slowdowns in the presence of main memory interference, and 2) use this model to devise two new memory scheduling techniques that either aim to satisfy slowdown bounds of applications or improve system fairness and performance. We already described Stall Time Fair Memory scheduling [28], Fairness via Source Throttling [6] and Du Bois et al.'s mechanism [5] for estimating an application's alone execution time *while* it is run alongside other applications in Section 2, and provided qualitative and quantitative comparison to STFM in Section 6. In this section, we present other related work.

Prior Work on Slowdown Estimation. Eyerman and Eeckhout [8] and Cazorla et al. [3] propose mechanisms to determine an application's slowdown while it is running alongside other applications on an SMT processor. Luque et al. [23] estimate application slowdowns in the presence of shared cache interference. These studies do not take into account inter-application interference at the main memory. Therefore, our proposed MISE model to estimate slowdown due to main memory interference can be favorably combined with the above approaches to quantify interference at the SMT processor and shared cache to build a comprehensive mechanism.

Prior Work on QoS. Several prior works have attempted to provide QoS guarantees in shared memory CMP systems. Mars et al. [24] propose a mechanism to estimate an application's sensitivity towards interference and its propensity to cause interference. They utilize this knowledge to make informed mapping decisions between applications and cores. However, this mechanism 1) assumes *a priori* knowledge of applications, which may not always be possible to have, and 2) is designed for only 2 cores, and it is not clear how it can be extended to more than 2 cores. In contrast, MISE does not assume any *a priori* knowledge of applications and works well with large core counts, as we have shown in this paper. That said, MISE can possibly be used to provide feedback to the mapping mechanism proposed by [24] to overcome the shortcomings of their mechanism. Iyer et al. [11, 14, 15] proposed mechanisms to provide guarantees on shared cache space, memory bandwidth or IPC for different applications. The slowdown guarantee provided by MISE-QoS is stricter than these mechanisms as MISE-QoS takes into account the alone-performance of each application. Nesbit et al. [30] proposed a mechanism to enforce a bandwidth allocation policy – partition the available bandwidth across concur-

rently running applications based on some policy. While we use a scheduling technique similar to lottery-scheduling [32, 36] to enforce the bandwidth allocation policies of MISE-QoS and MISE-Fair, the mechanism proposed by Nesbit et al. can also be used in our proposal to allocate bandwidth instead of our lottery-scheduling approach.

Prior Work on Memory Interference Mitigation. Much prior work has focused on the problem of mitigating inter-application interference at the main memory to improve system performance and/or fairness. Most of the previous approaches address this problem by modifying the memory request scheduling algorithm [2, 7, 13, 19, 20, 28, 29, 30]. We quantitatively compare MISE-Fair to STFM [28], ATLAS [19], and TCM [20], and show that MISE-Fair provides better fairness than these prior approaches. Prior work has also examined source throttling [6], memory channel/bank partitioning [16, 27], and memory interleaving [18] techniques to mitigate inter-application interference. These approaches are complementary to our proposed mechanisms and can be combined to achieve better fairness.

Prior Work on Analytical Performance Modeling. Several previous works [9, 10, 17, 35] have proposed analytical models to estimate processor performance, as an alternative to time consuming simulations. The goal of our MISE model, in contrast, is to estimate slowdowns at runtime, in order to enable mechanisms to provide QoS and high fairness. Its use in simulation is possible, but is left to future work.

10. Conclusion

We introduce a new, simple model, MISE, to estimate application slowdowns due to inter-application interference in main memory. MISE is based on two simple observations: 1) the rate at which an application's memory requests are served can be used as a proxy for the application's performance, and 2) the uninterfered request-service-rate of an application can be accurately estimated by giving the application's requests the highest priority in accessing main memory. Compared to the state-of-the-art approach for estimating main memory slowdowns, Stall-Time Fair Memory scheduling [28], MISE is simpler and more accurate, as our evaluations show.

We develop two new main memory request scheduling mechanisms that use MISE to achieve two different goals: 1) MISE-QoS aims to provide soft QoS guarantees to one or more applications of interest while ensuring high system performance, 2) MISE-Fair attempts to minimize maximum slowdown to improve overall system fairness. Our extensive evaluations show that our proposed mechanisms are more effective than the state-of-the-art memory scheduling approaches [15, 19, 20, 28] in achieving their respective goals.

We conclude that MISE is a promising substrate to build effective mechanisms that can enable the design of more predictable and more controllable systems where main memory is shared between multiple workloads. In future, we aim to devise similar simple models for accurately estimating application slowdowns in other shared resources.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback and suggestions. We acknowledge members of the SAFARI group for their feedback and for the stimulating research environment they provide. Many thanks to Brian Prasky from IBM and Arup Chakraborty from Freescale for their helpful comments. We acknowledge the support of our industrial sponsors, including AMD, HP Labs, IBM, Intel, Oracle, Qualcomm and Samsung. This research was also partially supported by grants from NSF (CAREER Award CCF-0953246), SRC and Intel URO Memory Hierarchy Program.

References

[1] *SPEC CPU2006*. http://www.spec.org/spec2006.
[2] R. Ausavarungnirun et al. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *ISCA*, 2012.
[3] F. J. Cazorla et al. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE TC*, Jul. 2006.
[4] R. Das et al. Application-aware prioritization mechanisms for on-chip networks. In *MICRO*, 2009.
[5] K. Du Bois et al. Per-thread cycle accounting in multicore processors. In *HiPEAC*, 2013.
[6] E. Ebrahimi et al. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *ASPLOS*, 2010.
[7] E. Ebrahimi et al. Parallel application memory scheduling. In *MICRO*, 2011.
[8] S. Eyerman and L. Eeckhout. Per-thread cycle accounting in SMT processors. In *ASPLOS*, 2009.
[9] S. Eyerman et al. A performance counter architecture for computing accurate CPI components. In *ASPLOS*, 2006.
[10] S. Eyerman et al. A mechanistic performance model for superscalar out-of-order processors. *ACM TOCS*, May 2009.
[11] A. Herdrich et al. Rate-based QoS techniques for cache/memory in CMP platforms. In *ICS*, 2009.
[12] Intel. First the tick, now the tock: Next generation Intel microarchitecure (Nehalem). *Intel Technical White Paper*, 2008.
[13] E. Ipek et al. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.
[14] R. Iyer. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *ICS*, 2004.
[15] R. Iyer et al. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, 2007.
[16] M. K. Jeong et al. Balancing DRAM locality and parallelism in shared memory CMP systems. In *HPCA*, 2012.
[17] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA*, 2004.
[18] D. Kaseridis et al. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *MICRO*, 2011.
[19] Y. Kim et al. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*, 2010.
[20] Y. Kim et al. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *MICRO*, 2010.
[21] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.
[22] K. Luo et al. Balancing thoughput and fairness in SMT processors. In *ISPASS*, 2001.
[23] C. Luque et al. CPU accounting in CMP processors. *IEEE CAL*, Jan. - Jun. 2009.
[24] J. Mars et al. Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, 2011.
[25] Micron. 4Gb DDR3 SDRAM.
[26] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security*, 2007.
[27] S. P. Muralidhara et al. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *MICRO*, 2011.
[28] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO*, 2007.
[29] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA*, 2008.
[30] K. J. Nesbit et al. Fair queuing memory systems. In *MICRO*, 2006.
[31] H. Patil et al. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.
[32] D. Petrou et al. Implementing lottery scheduling: Matching the specializations in traditional schedulers. In *USENIX ATEC*, 1999.
[33] S. Rixner et al. Memory access scheduling. In *ISCA*, 2000.
[34] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS*, 2000.
[35] K. Van Craeynest et al. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *ISCA*, 2012.
[36] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *OSDI*, 1994.
[37] W. K. Zuravleff and T. Robinson. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. Patent 5630096, 1997.