The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory

Lavanya Subramanian*§ Vivek Seshadri* Arnab Ghosh*†
Samira Khan*‡ Onur Mutlu*

*Carnegie Mellon University §Intel Labs †IIT Kanpur ‡University of Virginia

Abstract

In a multi-core system, interference at shared resources (such as caches and main memory) slows down applications running on different cores. Accurately estimating the slowdown of each application has several benefits: e.g., it can enable shared resource allocation in a manner that avoids unfair application slowdowns or provides slowdown guarantees. Unfortunately, prior works on estimating slowdowns either lead to inaccurate estimates, do not take into account shared caches, or rely on a priori application knowledge. This severely limits their applicability.

In this work, we propose the *Application Slowdown Model (ASM)*, a new technique that accurately estimates application slowdowns due to interference at both the shared cache and main memory, in the absence of a priori application knowledge. ASM is based on the observation that the performance of each application is strongly correlated with the rate at which the application accesses the shared cache. Thus, ASM reduces the problem of estimating slowdown to that of estimating the *shared cache access rate* of the application had it been run alone on the system. To estimate this for each application, ASM periodically *1) minimizes interference for the application at the main memory, 2) quantifies the interference the application receives at the shared cache*, in an aggregate manner for a large set of requests. Our evaluations across 100 workloads show that ASM has an average slowdown estimation error of only 9.9%, a 2.97x improvement over the best previous mechanism.

We present several use cases of ASM that leverage its slowdown estimates to improve fairness, performance and provide slowdown guarantees. We provide detailed evaluations of three such use cases: slowdown-aware cache partitioning, slowdown-aware memory bandwidth partitioning and an example scheme to provide soft slowdown guarantees. Our evaluations show that these new schemes perform significantly better than state-of-the-art cache partitioning and memory scheduling schemes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
MICRO-48, December 05-09, 2015, Waikiki, HI, USA
©2015 ACM. ISBN 978-1-4503-4034-2/15/12 $15.00
DOI: http://dx.doi.org/10.1145/2830772.2830803

1. Introduction

In most multi-core systems, applications running on different cores share the last-level cache and main memory. These applications contend for the shared resources, causing interference to each other. As a result, applications are slowed down compared to when they are run alone on the system. Since different applications have different sensitivity to shared resources, the slowdown of an application depends heavily on co-running applications and the available shared resources.

The ability to accurately estimate application slowdowns can enable several useful mechanisms. For instance, estimating the slowdown of each application may enable a cloud service provider to estimate the impact of interference on each application from consolidation on shared hardware resources, thereby billing the users appropriately [1, 2]. Perhaps more importantly, accurate slowdown estimates may enable allocation of shared resources to different applications in a slowdown-aware manner, thereby satisfying the performance requirements of different applications.

The slowdown of an application is the ratio of the execution time of the application when it is run with other applications (*shared execution time*) and the execution time of the application had it been run alone on the same system (*alone execution time*). While an application's shared execution time can be directly measured *while* it is running with other applications, the key question is how to estimate an application's *alone* execution time. Some previous works proposed techniques to estimate applications' alone execution times by profiling applications offline (e.g., [16, 17, 18, 40]). However, in many scenarios, such offline profiling of applications might not be feasible or accurate. For instance, in a cloud service, where any user can run a job using the available resources, profiling every application offline to gain a priori application knowledge can be prohibitive. Similarly, in a mobile environment, where multiple foreground and background jobs are run together, it is likely not possible to get a profile of the applications a priori. Finally, regardless of the usage case of a system, if the resource usage of an application is heavily input set dependent, the profile may not be representative. *Therefore, a key challenge in estimating the slowdown of an application is estimating its alone execution time online, i.e., at runtime, without actually running the application alone.*

While several prior works (e.g., [14, 15, 46, 66]) proposed online mechanisms to estimate application slowdowns, these mechanisms are either inaccurate and/or do not

take into account shared caches. Fairness via Source Throttling (FST) [15] and Per-thread cycle accounting (PTCA) [14] estimate slowdown due to *both* shared cache and main memory interference. However, their estimates have high inaccuracy. The key shortcoming of FST and PTCA is that they determine the effect of interference on slowdown at a *per-request granularity*. Specifically, to estimate an application's alone execution time, both FST and PTCA determine the number of cycles by which *each request* of the application is delayed due to interference at the shared cache and main memory. The drawback of this approach is that with the abundant amounts of parallelism in the memory subsystem, the service of different requests will likely overlap. As a result, estimating the effect of interference on slowdown at an *individual request granularity* is difficult and leads to inaccurate estimates for FST and PTCA, as we show in this paper. With a shared cache, the problem is exacerbated since the overlap behavior of memory requests could be *very different* when an application shares the cache with other applications as opposed to when it is run alone.

Our goal is to develop an *online* model to *accurately* estimate the slowdown of each application due to *both shared cache and main memory interference*. To this end, we propose the Application Slowdown Model (ASM). In contrast to prior approaches, ASM estimates an application's slowdown by observing its *aggregate request service behavior* rather than *individual* request service behavior to quantify interference. ASM does so by exploiting a new observation: the performance of each application is roughly proportional to the rate at which it accesses the shared cache. This observation enables us to estimate slowdown as the ratio of the cache access rate when the application is running alone ($\mathit{Cache\text{-}Access\text{-}Rate}_{\text{alone}}$ or CAR_{alone}) and the cache access rate when the application is running together with other applications ($\mathit{Cache\text{-}Access\text{-}Rate}_{\text{shared}}$ or CAR_{shared}).[1] CAR_{shared} can be directly measured when the application is running with other applications. The key challenge with our approach is how to accurately estimate CAR_{alone}.

To address this challenge, ASM *periodically* estimates CAR_{alone} for an application, by employing two key steps that 1) minimize the impact of memory bandwidth contention and 2) quantify the impact of shared cache capacity contention. First, in order to factor out the impact of memory bandwidth contention on CAR_{alone}, ASM *minimizes interference for an application at the main memory* by giving the application's requests the highest priority at the memory controller (similar to [66]). Doing so also enables ASM to get an accurate estimate of the *average cache miss service time* of the application had it been run alone (to be used in the next step). Second, *ASM quantifies the effect of interference at the shared cache* by using an auxiliary tag store [53, 56] to determine the number of shared cache misses that would have been hits if the application did not share the cache with other applications (these are called *contention misses*). This aggregate contention miss count is used along with the average miss service time (from the previous step) to estimate the time it would have taken to serve the application's requests had it been run alone.[2] This, in turn, gives us a good estimate of the application's CAR_{alone}.

In essence, one key contribution of our work is a new model that estimates an application's slowdown in the presence of interference by using previously-proposed structures (e.g., auxiliary tag stores and memory controller prioritization mechanisms) in a novel way: instead of trying to estimate the interference-caused delay of each request, we use these structures to estimate the aggregate reduction in cache access rates of interfering applications.

We evaluate the accuracy of ASM comparatively to two state-of-the-art models. Our evaluations using 100 4-core workloads show that ASM's slowdown estimates are much more accurate, achieving an average error of only 9.9% compared to the 2.97x higher average error of 29.4% for FST [15], the best previous model.

We present four use cases that *leverage ASM's slowdown estimates* to achieve different goals: 1) a new slowdown-aware cache partitioning scheme, 2) a new slowdown-aware memory bandwidth partitioning scheme, 3) a new scheme to provide soft slowdown guarantees, 4) mechanisms for fair pricing in a cloud service. We present detailed evaluations of the first two and a brief evaluation of the third. Our evaluations demonstrate significant fairness improvement, while maintaining and sometimes also improving performance, compared to state-of-the-art cache partitioning (Utility-based Cache Partitioning [56], memory-level parallelism and cache-friendliness aware partitioning [27]) and memory scheduling (Thread-Cluster Memory Scheduling [31], Parallelism-aware Batch Scheduling [47]) schemes. We also demonstrate that ASM can provide slowdown guarantees in the presence of shared cache and memory interference. Section 7 presents these use cases and evaluations.

This paper makes the following contributions:

- We introduce the Application Slowdown Model (ASM), an *online* model that *accurately* estimates application slowdowns due to *both* shared cache capacity and memory bandwidth interference.
- We compare ASM to two state-of-the-art models across a wide range of workloads and system configurations, showing that ASM's slowdown estimates are significantly more accurate than previous models' estimates.
- We present four use cases that leverage slowdown estimates from ASM to achieve different goals such as improving fairness and providing slowdown guarantees.

[1] In contrast, prior works [14, 15, 46] estimate an application's slowdown as a ratio of execution times (i.e., *shared-execution-time/alone-execution-time*), which entails estimating the performance impact of the delay of *every individual request* of the application and removing this estimated impact from *shared-execution-time* in order to get an estimate for the *alone-execution-time*.

[2] Note that ASM's use of auxiliary tag stores [53, 56] is different from previous works [14, 15], which also used them, but to estimate *per-request* as opposed to *aggregate* interference impact.

- We quantitatively evaluate three of these use cases that leverage ASM to partition the shared cache and memory bandwidth in order to improve fairness and provide slowdown guarantees. Our mechanisms significantly improve fairness, and sometimes also performance, compared to four state-of-the-art cache partitioning and memory scheduling approaches and provide slowdown guarantees.

2. Background and Motivation

A modern system typically consists of multiple levels of private caches, a shared last-level cache and off-chip main memory. In such a system, there are two main sources of inter-application interference: 1) shared cache capacity, and 2) main memory bandwidth. First, due to shared cache capacity contention, an application's accesses that would otherwise have hit in the cache (had the application been run alone), *miss* in the cache, increasing average memory access latency. Second, requests from different applications interfere at the different main memory components, such as buses, banks and row-buffers [43, 46, 47]. As a result, applications delay each other's requests due to conflicts at these different components.

Contention at both the shared cache and main memory increases the overall memory access latency, significantly slowing down different applications [43, 46, 47]. Moreover, an application's slowdown depends on the sensitivity of its performance to cache capacity and memory bandwidth, making *online slowdown estimation* a hard problem.

2.1. Previous Work on Slowdown Estimation

Several prior works have attempted to estimate application slowdown due to shared cache capacity and/or memory bandwidth interference (e.g., [14, 15, 43, 46, 66]). As we mentioned in Section 1, the key challenge in estimating slowdown is how to accurately estimate the performance of the application had it been running alone, without actually running it alone.

Stall-Time Fair Memory (STFM) scheduler [46] estimates an application's slowdown as the ratio of its alone and shared main memory stall times. To estimate the alone memory stall time for an application, STFM counts the number of cycles each request of the application is stalled due to interference at the buses, banks, and the row-buffers. Unfortunately, due to the abundant parallelism present in main memory, service of different requests could overlap significantly, making it difficult to accurately estimate the interference cycles for each application. As a result, STFM's slowdown estimates are inaccurate [66]. Memory-interference Induced Slowdown Estimation (MISE) [66] addresses this problem using the observation that an application's performance is correlated with its memory request service rate and estimates slowdown as the ratio of request service rates. To measure the alone request service rate of an application, MISE periodically gives the application's requests the highest priority in accessing main memory. The main drawback of both STFM and MISE is that they do not account for shared cache interference.

Fairness via Source Throttling (FST) [15] and Per-thread cycle accounting (PTCA) [14] estimate application slowdowns due to both shared cache capacity and main memory bandwidth interference. They compute slowdown as the ratio of alone and shared execution times and estimate alone execution time by determining the number of cycles by which each request is delayed. Both FST and PTCA use a mechanism similar to STFM to quantify interference at the main memory. To quantify interference at the shared cache, both mechanisms determine which accesses of an application miss in the shared cache but would have been hits had the application been run alone on the system (contention misses), and compute the number of additional cycles taken to serve *each contention miss*. The main difference between FST and PTCA is in the mechanism they use to identify a contention miss. FST uses a *pollution filter* for each application that tracks the blocks of the application that were evicted by other applications. Any access that misses in the cache and hits in the pollution filter is considered a contention miss. On the other hand, PTCA uses an *auxiliary tag store* [53, 56] for each application that tracks the expected state of the cache had the application been running alone on the system. PTCA classifies any access that misses in the cache and hits in the auxiliary tag store as a contention miss.[3]

2.2. Motivation and Our Goal

In this work, we find that the approach used by FST and PTCA results in significantly inaccurate slowdown estimates due to two main reasons. First, both FST and PTCA quantify interference at an *individual request granularity*. Given the abundant parallelism in the memory subsystem, service of different requests overlap significantly [22, 48]. As a result, accurately estimating the number of cycles by which each request is delayed due to interference is difficult. In fact, STFM [46] recognizes this problem and introduces a *parallelism factor* as a fudge factor. A shared cache only makes the problem worse as the request stream of an application to main memory could be *completely different* when it shares the cache with other applications versus when it runs alone.

Second, the mechanisms used by FST and PTCA to identify contention misses, i.e., pollution filters or auxiliary tag stores, incur high hardware overhead. In order to reduce this overhead, the pollution filters are made approximate [8, 15, 63] and the auxiliary tag stores are sampled [14, 54, 55]. For example, PTCA maintains the auxiliary tag store only for a few sampled cache sets. The interference cycles for the requests that map to the sampled sets are counted and scaled accordingly to account for the total interference cycle count for all requests. While such techniques reduce hardware cost, they dras-

[3] Later, in Sections 3.2 and 3.3, we will describe how our mechanism differs from these mechanisms in estimating contention misses.

tically reduce the accuracy of the slowdown estimates (as we show quantitatively in Section 6), since the interference behavior varies widely across requests and is difficult to estimate accurately by sampling and scaling.

Such inaccuracies in slowdown estimates from prior works severely limit their applicability. **Our goal** is to overcome these shortcomings and build an *accurate, low-overhead, online* slowdown estimation model that takes into account interference at *both* the shared cache and main memory. To this end, we propose our *Application Slowdown Model (ASM)*. In the next section, we describe the key ideas and challenges in building our model.

3. Overview of ASM

In contrast to prior works, which quantify interference at a per-request granularity, ASM uses *aggregate request behavior* to quantify interference, based on the following observation.

3.1. Observation: Access Rate as a Proxy for Performance

The performance of each application is proportional to the rate at which it accesses the shared cache.

Intuitively, an application can make progress when its data accesses are served. The faster its accesses are served, the faster it makes progress. In the steady state, the rate at which an application's accesses are served (service rate) is almost the same as the rate at which it generates accesses (access rate). Therefore, if an application can generate more accesses to the cache in a given period of time (higher access rate), then it can make more progress during that time (higher performance).

In fact, MISE [66] observes that the performance of a *memory-bound application* is proportional to the rate at which its memory accesses are served. Our observation is stronger than MISE's observation because our observation relates performance to the shared cache access rate and not just main memory access rate, thereby accounting for the impact of both shared cache and main memory interference. Hence, it holds for a broader class of applications that are sensitive to cache capacity and/or memory bandwidth, and not just memory-bound applications.

To validate our observation, we conducted an experiment in which we run each application of interest alongside a memory bandwidth/cache capacity hog program on an Intel Core-i5 processor with a 6MB shared cache. The cache and memory access behavior of the hog can be varied to cause different amounts of interference to the main program. Each application is run multiple times with the hog with different characteristics. During each run, we measure the performance and shared cache access rate of the application.

Figure 1 plots the results of our experiment for three applications from the SPEC CPU2006 suite [4]. The plot shows cache access rate vs. performance of the application normalized to when it is run alone. As our results indicate, the performance of each application is indeed proportional to the cache access rate of the application, validating our observation. We observed the same behavior for a wide range of applications.

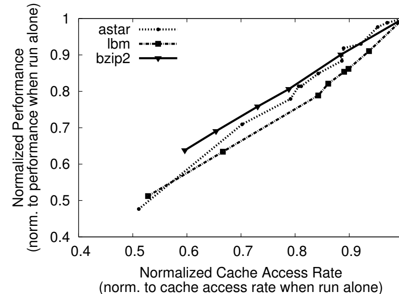

Figure 1: Cache access rate vs. performance

ASM exploits our observation to estimate slowdown as a ratio of cache access rates, instead of as a ratio of performance.

$$\textit{performance} \propto \textit{cache-access-rate}\ (CAR)$$

$$\text{Slowdown} = \frac{\textit{performance}_{\text{alone}}}{\textit{performance}_{\text{shared}}} = \frac{CAR_{\text{alone}}}{CAR_{\text{shared}}}$$

While CAR_{shared}, which is a proxy for $\textit{performance}_{\text{shared}}$ is easy to measure, the challenge is in estimating CAR_{alone}, which is a proxy for $\textit{performance}_{\text{alone}}$.

CAR_{alone} vs. $\textit{performance}_{\text{alone}}$. In order to estimate an application's slowdown during a given interval, prior works estimate its alone execution time ($\textit{performance}_{\text{alone}}$) by tracking the interference experienced by *each of the application's requests served during this interval* and subtracting these interference cycles from the application's shared execution time ($\textit{performance}_{\text{shared}}$). This approach leads to inaccuracy, since estimating per-request interference is difficult due to the parallelism in the memory system (Section 2.2). CAR_{alone}, on the other hand, can be estimated more accurately by exploiting the observation made by several prior works that applications' phase behavior is relatively stable over time scales on the order of a few million cycles (e.g., [24, 62]). Hence, we propose to estimate CAR_{alone} periodically over short time periods, during which 1) the memory bandwidth interference experienced by an application is minimized and 2) shared cache capacity interference experienced by the application is quantified. We describe these mechanisms and their benefits in detail in the next sections.

3.2. Challenge: Accurately Estimating CAR_{alone}

A naive way of estimating CAR_{alone} of an application periodically is to run the application by itself for short periods of time and measure CAR_{alone}. While such a scheme would eliminate main memory interference, it would *not* eliminate shared cache interference, since the caches cannot be warmed up at will in a short time duration. Hence, it is not possible to take this approach to estimate CAR_{alone} accurately (at least with low performance overhead). Therefore, ASM takes a hybrid approach to estimate CAR_{alone} for each application by

1) minimizing interference at the main memory, and 2) quantifying interference at the shared cache.

Step 1: Minimizing main memory interference. One way of minimizing main memory interference is to periodically give *all* the memory bandwidth to each application, in turn, for short time periods. However, such a scheme would not only waste memory bandwidth but also likely be inaccurate if the corresponding application did not generate many memory requests during its period. Therefore, ASM minimizes interference for each application at the main memory by simply giving each application's requests *the highest priority* at the memory controller periodically for short lengths of time (as proposed by previous work [66]). This step results in two key outcomes. First, it eliminates most of the impact of main memory interference when ASM is estimating CAR_{alone} for the application (Section 4.3 describes how we account for the remaining minimal interference). Second, it provides ASM with an accurate estimate of the cache miss service time for the application in the absence of main memory interference. This estimate is used in the next step, in quantifying shared cache interference for the application. Furthermore, it is important to note that each application is given highest priority *for only short time periods*, thereby preventing it from causing interference to other applications for a long time (this leads to negligible change in performance/fairness compared to the baseline: performance degrades by ~1% and fairness improves by ~1%).

Step 2: Quantifying shared cache interference. To quantify the effect of cache interference on the application, we need to identify the excess cycles that are spent for serving shared cache misses that are *contention misses*—those that would have otherwise hit in the cache had the application run alone on the system. We use an auxiliary tag store [53, 56] for each application to first identify contention misses. Once we determine the aggregate number of contention misses, we use the average cache miss service time (computed in the previous step) and average cache hit service time (periodically computed as the average number of cycles taken to serve cache hits) to estimate the excess number of cycles spent serving the contention misses—essentially quantifying the effect of shared cache interference.

Overall model for CAR_{alone}. In summary, ASM estimates CAR_{alone} by 1) minimizing interference at the main memory and 2) quantifying interference at the shared cache, for each application. In the first step, in order to minimize interference at the main memory for an application, the application is given highest priority at the memory controller for short periods of time. This 1) eliminates most of the impact of main memory interference when estimating CAR_{alone} for the application and 2) provides ASM with an accurate estimate of the cache miss service time of the application. In the second step, to quantify interference at the shared cache for the application, the miss service time estimates from the first step are used along with contention miss counts from an auxiliary tag store to estimate the excess cycles spent serving contention misses, when an application is given highest priority at the memory controller. These excess cycles, which would not have been experienced by the application had it run alone, are then removed from the number of cycles during which each application is given highest priority, to calculate CAR_{alone}. We describe the details of estimating CAR_{alone} in Section 4.

3.3. Why is ASM Better than Prior Work?

ASM is better than prior work due to three reasons. First, as we describe in Section 2.2, prior works aim to estimate the effect of main memory interference on each request individually, which is difficult and inaccurate. In contrast, our approach eliminates most of the main memory interference for an application by giving the application's requests the highest priority (step 1), which also allows ASM to gather a good estimate of the *average cache miss service time*. Second, to quantify the effect of shared cache interference, ASM only needs to identify the *number* of contention misses (step 2), unlike prior approaches that need to determine how long each contention miss is delayed due to interference. This makes ASM more amenable to hardware-overhead-reduction techniques like set sampling (see Section 4.4 for details). In other words, the error introduced by set sampling in estimating the number of contention misses is *far lower* than the error it introduces in estimating *the actual number of cycles by which each contention miss is delayed due to interference*. Third, as we describe in Section 7.1, ASM enables the estimation of slowdowns for *different cache allocations* in a straightforward manner, which is non-trivial using prior models.

4. Implementing ASM

Applications have multiple phases. As a result, the slowdown of each application due to shared cache and main memory interference could vary with time. To account for this, ASM divides execution into multiple quanta, each of length Q cycles (a few million cycles; see Section 6.6). At the end of each quantum, ASM 1) measures CAR_{shared}, and 2) estimates CAR_{alone} for each application, and reports the slowdown of each application as the ratio of the application's CAR_{alone} and CAR_{shared}.

4.1. Measuring CAR_{shared}

Measuring CAR_{shared} for each application is fairly straightforward. ASM keeps a per-application counter that tracks the number of shared cache accesses for the application. The counter is cleared at the beginning of each quantum and is incremented whenever there is a new shared cache access for the application. At the end of each quantum, the CAR_{shared} for each application can be computed as

$$\textit{Cache-Access-Rate}_{\text{shared}} = \frac{\text{\# Shared Cache Accesses}}{Q}$$

4.2. Estimating CAR_{alone}

As we described in Section 3.2, during each quantum, ASM periodically estimates the CAR_{alone} of each application by 1) minimizing interference at the main memory and 2) quantifying interference at the shared cache. Towards this end, ASM divides each quantum into epochs of length E cycles (thousands of cycles), similar to [66]. Each epoch is probabilistically assigned to one of the co-running applications. During each epoch, ASM collects information for the corresponding application that is later used to estimate CAR_{alone} for the application. Each application has equal probability of being assigned an epoch. Assigning epochs to applications in a round-robin fashion could also achieve similar effects. However, we build probabilistic mechanisms on top of ASM that allocate bandwidth to applications in a slowdown-aware manner (Section 7.2). In order to facilitate building such mechanisms on top of ASM, we employ a policy that probabilistically assigns an application to each epoch.

At the beginning of each epoch, ASM communicates the ID of the application assigned to the epoch to the memory controller. During that epoch, the memory controller gives the corresponding application's requests the highest priority in accessing main memory.

To track contention misses, ASM maintains an auxiliary tag store [53,56] for each application that tracks the state of the cache had the application been running alone. In this section, we will assume a full auxiliary tag store for ease of description. However, as we will describe in Section 4.4, our final implementation uses set sampling [53,56] to significantly reduce the overhead of the auxiliary tag store with negligible loss in accuracy.

Table 1 lists the metrics measured by ASM for each application during the epochs that are assigned to it. At the end of each quantum, ASM uses these metrics to estimate the CAR_{alone} of the application. Each metric is measured using a counter while the application is running with other applications.

Metric	Definition
epoch-count	# epochs assigned to the application
epoch-hits	Total # shared cache hits for the application during its assigned epochs
epoch-misses	Total # shared cache misses for the application during its assigned epochs
epoch-hit-time	# cycles during which the application has at least one outstanding hit during its assigned epochs
epoch-miss-time	# cycles during which the application has at least one outstanding miss during its assigned epochs
epoch-ATS-hits	# auxiliary tag store hits for the application during its assigned epochs
epoch-ATS-misses	# auxiliary tag store misses for the application during its assigned epochs

Table 1: Metrics measured by ASM for each application to estimate CAR_{alone}

The CAR_{alone} of an application is given by:

$$CAR_{\text{alone}} = \frac{\#\text{ Requests during application's epochs}}{\text{Time to serve requests when run alone}} = \frac{\textit{epoch-hits} + \textit{epoch-misses}}{(\textit{epoch-count} * E) - \textit{epoch-excess-cycles}}$$

where, *epoch-count* $* E$ represents the actual time the system spent prioritizing requests from the application, and *epoch-excess-cycles* is the number of excess cycles spent serving the application's contention misses—those that would have been hits had the application run alone.

At a high level, for each contention miss, the system spends the time of serving a miss as opposed to a hit had the application been running alone. Therefore,

$$\textit{epoch-excess-cycles} = (\#\text{ Contention Misses}) \times (\textit{avg-miss-time} - \textit{avg-hit-time})$$

where, *avg-miss-time* is the average miss service time and *avg-hit-time* is the average hit service time for the application for requests served during all of the application's epochs. Each of these terms is computed using the metrics measured by ASM, as follows.

$$\#\text{ Contention Misses} = \textit{epoch-ATS-hits} - \textit{epoch-hits}$$

$$\textit{avg-miss-time} = \frac{\textit{epoch-miss-time}}{\textit{epoch-misses}}$$

$$\textit{avg-hit-time} = \frac{\textit{epoch-hit-time}}{\textit{epoch-hits}}$$

4.3. Accounting for Memory Queueing Delay

During each epoch, when there are *no* requests from the highest priority application, the memory controller may schedule requests from *other* applications. If a high priority request arrives after *another* application's request is scheduled, it may be delayed. To address this problem, we apply a mechanism proposed by prior work [66], wherein ASM measures the number of *queueing cycles* for each application using a counter. A cycle is deemed a queueing cycle if a request from the highest priority application is outstanding and the previous command issued by the memory controller was from *another* application. At the end of each quantum, the counter represents the queueing delay for all *epoch-misses*. However, since ASM has already accounted for the queueing delay of the contention misses during its previous estimate by removing the *epoch-excess-cycles* taken to serve contention misses, it only needs to account for the queueing delay for the remaining misses that would have occurred even if the application were running alone, i.e., *epoch-ATS-misses*. To do this, ASM computes the average queueing cycles for each miss from the application:

$$\textit{avg-queueing-delay} = \frac{\#\text{ queueing cycles}}{\textit{epoch-misses}}$$

and computes its **final CAR_{alone} estimate** as

$$CAR_{\text{alone}} = \frac{\textit{epoch-hits} + \textit{epoch-misses}}{(\textit{epoch-count} * E) - \textit{epoch-excess-cycles} - (\textit{epoch-ATS-misses} * \textit{avg-queueing-delay})}$$

4.4. Sampling the Auxiliary Tag Store

As we mentioned before, in our final implementation, we use set sampling [53, 56] to reduce the overhead of the auxiliary tag store (ATS). Using this approach, the ATS is maintained only for a small number of sampled sets. The only two quantities that are affected by sampling are *epoch-ATS-hits* and *epoch-ATS-misses*. With sampling enabled, we first measure the fraction of hits/misses in the sampled ATS. We then compute *epoch-ATS-hits*/*epoch-ATS-misses* as a product of the hit/miss fraction with the total number of cache accesses.

$$\textit{epoch-ATS-hits} = \textit{ats-hit-fraction} \times \textit{epoch-accesses}$$
$$\textit{epoch-ATS-misses} = \textit{ats-miss-fraction} \times \textit{epoch-accesses}$$

where *epoch-accesses* = *epoch-hits* + *epoch-misses*.

4.5. Hardware Cost

ASM tracks the seven metrics in Table 1 and # *queueing cycles* using registers. We find that using a four byte register for each of these counters is sufficient for the values they track. Hence, the counter overhead is 32 bytes for each hardware context. In addition to these counters, an auxiliary tag store (ATS) is maintained for each hardware context. The ATS size depends on the number of sets that are sampled. For 64 sampled sets and 16 ways per set, assuming four bytes for each entry, the overhead is 4KB *per-hardware context*, which is 0.2% of the size of a 2MB shared cache (used in our main evaluations). Hence, for 4/8/16 core systems with one hardware context per-core, the overhead is 0.8%/1.6%/3.2% of the size of a 2MB shared cache.

5. Methodology

System Configuration. We model the main memory system using a cycle-level in-house DDR3-SDRAM simulator, similar to Ramulator [32]. We validated the simulator against DRAMSim2 [59] and Micron's behavioral Verilog model [41]. We integrate our DRAM simulator with an in-house simulator that models out-of-order cores with a Pin [37] frontend and PinPoints [52] to capture the representative phases of workloads. We model a per-core private L1 cache and a shared L2 cache. Table 2 lists the main system parameters. Our main evaluations use a 4-core system with 2MB shared cache and 1-channel main memory. We plan to make the simulator publicly available at https://github.com/CMU-SAFARI.

Workloads. For our multiprogrammed workloads, we use applications from the SPEC CPU2006 [4] and NAS Parallel Benchmark [3] suites (run single-threaded). We construct workloads with varying memory intensity, randomly choosing applications for each workload. We run each workload for 100 million cycles. We present results for 100 4-core, 100 8-core and 100 16-core workloads.

Metrics. We use average error to compare the accuracy of ASM and previously proposed models. We compute slowdown estimation error for each application, at the end of every quantum (Q), as the absolute value of

Processor	4-16 cores, 5.3GHz, 3-wide issue, 128-entry instruction window
L1 cache	64KB, private, 4-way associative, LRU, line size = 64B, latency = 1 cycle
Last-level cache	1MB-4MB, shared, 16-way associative, LRU, line size = 64B, latency = 20 cycles
Mem. controller	128-entry request buffer per controller, FR-FCFS [58, 74] scheduling policy
Main Memory	DDR3-1333 (10-10-10) [42], 1-4 channels, 1 rank/channel, 8 banks/rank, 8KB rows

Table 2: Configuration of the simulated system

$$\text{Error} = \frac{\text{Estimated Slowdown} - \text{Actual Slowdown}}{\text{Actual Slowdown}} \times 100\%$$

$$\text{Actual Slowdown} = \frac{IPC_{alone}}{IPC_{shared}}$$

We compute IPC_{alone} for the same amount of work completed in the alone run as that completed in the shared run for each quantum. For each application, we compute the average error across all quanta in a workload run and then compute the average across all occurrences of the application in all of our workloads.

Parameters. We compare ASM with two previous slowdown estimation models: Fairness via Source Throttling (FST) [15] and Per-Thread Cycle Accounting (PTCA) [14]. For ASM, we set the quantum length (Q) to 5,000,000 cycles and the epoch length (E) to 10,000 cycles. For ASM and PTCA, we present results both with sampled and unsampled auxiliary tag stores (ATS). For FST, we present results with various pollution filter sizes that match the size of the ATS. Section 6.6 evaluates the sensitivity of ASM to quantum and epoch lengths.

6. Evaluation of the Model

Figure 2 compares the average slowdown estimation error from FST, PTCA, and ASM, *with no sampling* in the auxiliary tag store for PTCA and ASM, and equal-overhead pollution filter for FST. Our implementations of FST and PTCA take into account both memory bandwidth interference and shared cache capacity interference at a per-request granularity. The benchmarks on the left are from the SPEC CPU2006 suite and those on the right are from the NAS benchmark suite. Benchmarks within each suite are sorted based on memory intensity increasing from left to right. Figure 3 presents the corresponding results with a sampled auxiliary tag store (64 cache sets) for PTCA and ASM, and an equal-size pollution filter for FST.

We draw three major conclusions. First, even without sampling, ASM has significantly lower slowdown estimation error (9%) compared to FST (18.5%) and PTCA (14.7%) This is because, as described in Section 2.2, prior works attempt to quantify the effect of interference on a per-request basis, which is inherently inaccurate given the abundant parallelism in the memory subsystem. ASM, in contrast, uses aggregate request behavior to quantify

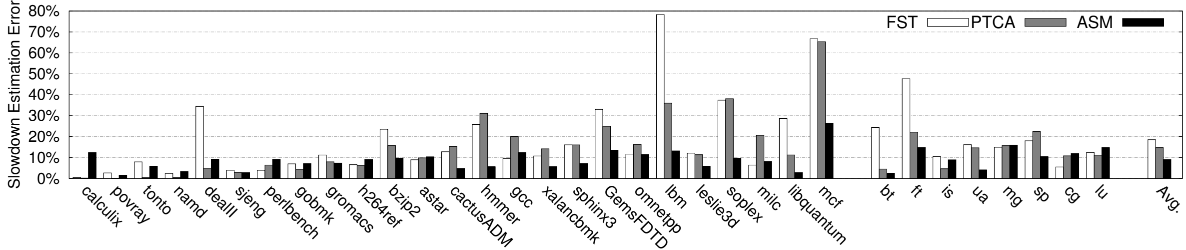

Figure 2: Slowdown estimation accuracy with no auxiliary tag store sampling

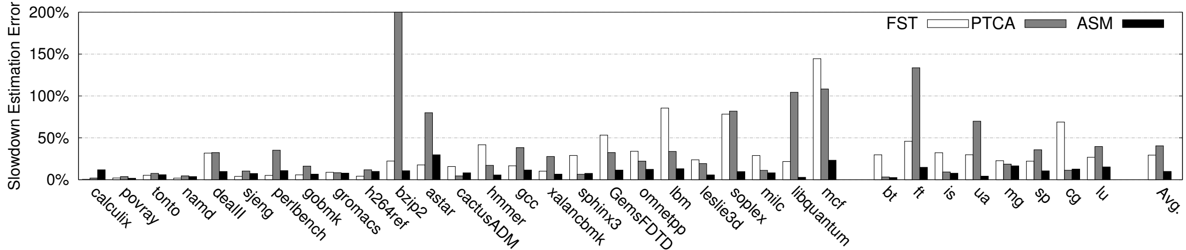

Figure 3: Slowdown estimation accuracy with auxiliary tag store sampling

the effect of interference, and hence is more accurate.

Second, sampling the auxiliary tag store and reducing the size of the pollution filter significantly increase PTCA and FST's estimation error respectively, while it has negligible impact on ASM's estimates. PTCA's error increases from 14.7% to 40.4% and FST's error increases from 18.5% to 29.4%, whereas ASM's error increases from 9% to only 9.9%. Again, as we describe in Section 2.2, PTCA's increase in error from sampling is because it estimates the number of cycles by which *each* contention miss (from the sampled sets) is delayed, and *scales up* this cycle count to the entire cache. However, since different requests experience different levels of interference, sampling introduces more error in the latency estimates of contention misses, as we show in Section 6.3, thereby introducing more error in PTCA's estimates. FST's slowdown estimation error also increases from sampling, but the increase is not as significant as PTCA's increase from sampling, because it uses a pollution filter that is implemented using a Bloom filter [8], which seems more robust to size reductions than an auxiliary tag store.

Third, FST and PTCA's slowdown estimates are particularly inaccurate for applications with high memory intensity (e.g., *soplex*, *libquantum*, *mcf*) and high cache sensitivity (e.g., *ft*, *dealII*, *bzip2*). This is because applications with high memory intensity generate a large number of requests to memory, and accurately modeling the overlap in service of such large number of requests individually is difficult, resulting in inaccurate slowdown estimates. Similarly, an application with high cache sensitivity is severely affected by shared cache interference. Hence, the request streams to main memory of the application are drastically different when the application is run alone versus when it shares the cache with other applications. This makes it hard to estimate interference on a per-request basis. ASM simplifies the problem by tracking aggregate behavior, resulting in significantly lower error for applications with high memory intensity and/or cache sensitivity.

In summary, with reasonable hardware overhead, ASM estimates slowdowns more accurately than prior work and is more robust to varying access behavior of applications.

Accuracy with Database Workloads. We evaluate FST, PTCA and ASM's slowdown estimation accuracy with database workloads - specifically, TPC-C [68] and the Yahoo Cloud Serving Benchmark (YCSB) [11]. The average error for FST (unsampled), PTCA (unsampled) and ASM (sampled) are 27%, 12% and 4% respectively. Hence, we conclude that ASM can effectively estimate slowdown in modern database workloads as well.

6.1. Distribution of Slowdown Estimation Error

Figure 4 shows the distribution of slowdown estimation error for FST, PTCA (both unsampled) and ASM (sampled), across all the 400 instances of different applications in our 100 4-core workloads. The x-axis shows error ranges and the y-axis shows what fraction of points lie in each range. Two observations are in order. First, 95.25% of ASM's estimates have an error less than 20%, whereas only 76.25% and 79.25% of FST and PTCA's estimates respectively have an error within 20%. Second, ASM's maximum error is only 36%, while FST/PTCA have maximum errors of 133%/87% respectively (not visible in the plot). We conclude that ASM's slowdown estimates have much lower variance than FST and PTCA's estimates and thus they are more robust.

6.2. Impact of Prefetching

Figure 5 shows the average slowdown estimation error for FST, PTCA and ASM, across 100 4-core workloads (unsampled), with a stride prefetcher [7, 63] of degree four and distance 24. The error bars show the standard deviation across workloads. ASM achieves a significantly low error of 7.5%, compared to 20% and 15% for FST and PTCA respectively. ASM's error reduces compared to not employing a prefetcher, since memory interference

induced stalls reduce with prefetching, which reduces the amount of interference whose impact on slowdowns needs to be estimated. This reduction in interference is true for FST and PTCA as well. However, their error increases slightly compared to not employing a prefetcher, since they estimate interference at a per-request granularity. The introduction of prefetches causes more disruption and hard-to-estimate overlap behavior among requests going to memory, making it more difficult to estimate interference at a per-request granularity. In contrast, ASM uses aggregate request behavior to estimate slowdowns, which is more robust, resulting in more accurate slowdown estimates with prefetching.

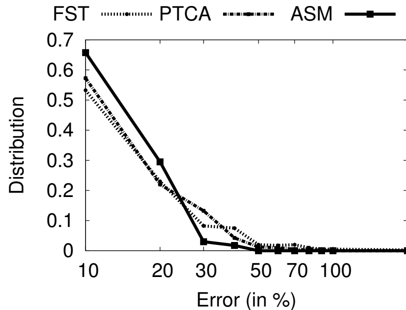

Figure 4: Error distribution

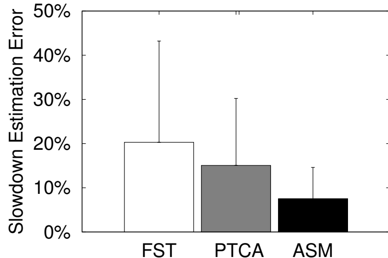

Figure 5: Prefetching impact

6.3. Latency Distribution: Benefits of Epoch-Based Aggregation

In order to provide more insight into the accuracy of FST, PTCA and ASM's slowdown estimates, Figures 6a and 6b present the distribution of the alone miss service times across 30 of our most memory-intensive workloads when actually measured and estimated with FST, PTCA and ASM, without and with sampling respectively. We make three major observations. First, even when no sampling is applied, FST and PTCA's estimated miss service time distributions are different from the actual measured distributions (particularly visible around 50 and 100 ns). This is because FST and PTCA rely on per-request latency estimation and hence, are not as effective in capturing request overlap behavior. Second, ASM, by virtue of estimating miss service time of an application across an aggregate set of requests when giving the application high priority, is able to estimate the miss service times much more accurately. Third, when the auxiliary tag store is sampled, both FST and PTCA's latency estimates (particularly PTCA's) deviate much more from actual measured miss service times. ASM, on the other hand, decouples the estimation of the number of contention misses and their miss service times, as describe in Section 4.2. Hence, its estimated miss service distribution remains mostly unchanged with sampling.

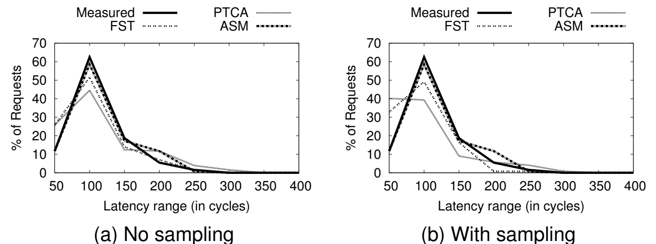

Figure 6: Miss service time distributions

6.4. Benefits of Estimating Shared Cache Interference

In this section, we seek to isolate the benefits of using epoch-based aggregation to account *solely* for memory interference from the benefits of using epoch-based aggregation for *both* main memory and shared cache interference. In order to do so, we evaluate the average slowdown estimation error of the MISE model [66] that estimates slowdown due *solely* to main memory interference and ASM that estimates slowdown due to *both* main memory and shared cache contention. Both MISE and ASM employ epoch-based aggregation. MISE has an average slowdown estimation error of 22%, across our 4-core workloads, whereas ASM has a much lower slowdown estimation error of 9.9%. This is because, MISE does not take into account shared cache interference, whereas ASM takes into account both main memory and shared cache interference. Hence, we conclude that employing epoch-based aggregation while accounting for *both* main memory and shared cache interference is key to achieving high accuracy.

6.5. Sensitivity to System Parameters

Core Count. Figure 7 presents sensitivity of slowdown estimates from FST, PTCA and ASM to core count. Since PTCA and FST's slowdown estimation errors degrade significantly with sampling, for our sensitivity studies, we present results for prior works with no sampling. However, for ASM, we still present results with a sampled auxiliary tag store. We evaluate 100 workloads for each core count. The error bars show standard deviation across all workloads.

We draw three conclusions. First, ASM's slowdown estimates are significantly more accurate than slowdown estimates from FST and PTCA across all core counts. Furthermore, the standard deviation of ASM's error is much lower than that of FST and PTCA, showing that its estimation error has a lower spread. Second, ASM's slowdown estimation accuracy reduces at higher core

counts. This is because of the residual interference from memory queueing despite giving applications high priority. We take this queueing into account, as we describe in Section 4.3. However, the effect of this queueing is greater at higher core counts and is harder to account for entirely. Third, ASM's accuracy gains over FST and PTCA increase with increasing core count. As core count increases, interference at the shared cache and main memory increases and, consequently, request behavior is even more different from alone run behavior. Hence, the slowdown estimation error increases for all models. However, ASM, by virtue of tracking aggregate request behavior has the least error increase compared to FST and PTCA.

Cache Capacity. Figure 8 shows the sensitivity of FST, PTCA and ASM's slowdown estimates to shared cache capacity, across all our 4-core workloads. ASM's slowdown estimates are significantly more accurate than FST and PTCA's estimates, across all cache capacities.

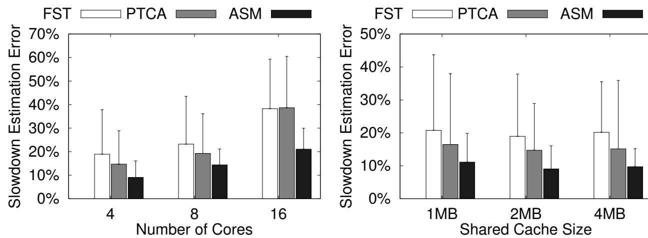

Figure 7: Error vs. core count **Figure 8: Error vs. cache size**

6.6. Sensitivity to Epoch and Quantum Lengths

Table 3 shows the average error, across all our workloads, for different values of the quantum (Q) and epoch lengths (E). As the table shows, the estimation error increases with decreasing quantum length and, in general, increasing epoch length. This is because the number of epochs (Q/E) decreases as quantum length (Q) decreases and/or epoch length (E) increases. With fewer epochs, some applications may not be assigned enough epochs to enable ASM to reliably estimate their CAR_{alone}. However, when the epoch length is very small, e.g., 1000 cycles, the estimation error is the highest. This is because when epochs are very short, applications are not prioritized for long enough periods of time at the memory controller, to emulate alone run behavior accurately. For our main evaluations, we use a quantum length of 5,000,000 cycles and epoch length of 10,000 cycles.

Quantum Length \ Epoch Length	1000	10000	50000	100000
1000000	18.4%	12%	14%	16.6%
5000000	17.1%	9.9%	10.6%	11.5%
10000000	16.9%	9.2%	9.9%	10.5%

Table 3: Error sensitivity to epoch and quantum lengths

7. Leveraging ASM

ASM's slowdown estimates can be leveraged to build various slowdown-aware mechanisms to improve performance, fairness, and provide slowdown guarantees. In this section, we present four such use cases of ASM.

7.1. ASM Cache Partitioning (ASM-Cache)

ASM-Cache partitions shared cache capacity among applications with the goal of minimizing slowdown. The key idea is to allocate more cache ways to applications whose slowdowns reduce the most from additional cache space.

7.1.1. Mechanism. ASM-Cache consists of two components. First, to partition the cache in a slowdown-aware manner, we estimate the slowdown of each application when it is given a different number of cache ways. Next, we determine the cache way allocation for each application based on the slowdown estimates, using a scheme similar to Utility-based Cache Partitioning [56].

Slowdown Estimation. We estimate slowdown of an application when it is allocated n ways as

$$slowdown_n = \frac{CAR_{\text{alone}}}{CAR_n}$$

where, CAR_n is the cache access rate of the application when n ways are allocated to it. We estimate CAR_{alone} using the mechanism described in Section 4. While CAR_n can be estimated by measuring it while giving all possible way allocations to each application, such an approach is expensive and detrimental to performance as the search space is huge. Therefore, we propose to estimate CAR_n using a mechanism similar to estimating CAR_{alone}.

Let *quantum-hits* and *quantum-misses* be the number of shared cache hits and misses for the application during a quantum. At the end of the quantum,

$$CAR_n = \frac{\textit{quantum-hits} + \textit{quantum-misses}}{\#\text{ Cycles to serve the above accesses with } n \text{ ways}}$$

The challenge is in estimating the denominator, i.e., the number of cycles taken to serve an application's shared cache accesses during the quantum, if the application had been given n ways. To estimate this, we first determine the number of shared cache accesses that would have hit in the cache had the application been given n ways ($\textit{quantum-hits}_n$). This can be directly obtained from the auxiliary tag store. (We use a sampling auxiliary tag store and scale up the sampled $\textit{quantum-hits}_n$ value using the mechanism described in Section 4.4.)

There are three cases: 1) $\textit{quantum-hits}_n = \textit{quantum-hits}$, 2) $\textit{quantum-hits}_n > \textit{quantum-hits}$, and 3) $\textit{quantum-hits}_n < \textit{quantum-hits}$. In the first case, when the number of hits with n ways is same as the number of hits during the quantum, we expect the system to take the same number of cycles to serve the requests even with n ways, i.e., Q cycles. In the second case, when there are more hits with n ways, we expect the system to serve the requests in fewer than Q cycles. Finally, in the third case, when there are fewer hits with n ways, we expect the system to take more than Q cycles to serve the requests. Let $\Delta hits$ denote $\textit{quantum-hits}_n - \textit{quantum-hits}$. If *quantum-hit-time* and *quantum-miss-time* are the average cache hit and miss service times respectively for the accesses of the

application during the quantum, we estimate the number of cycles to serve the requests with n ways as,

$$cycles_n = \mathrm{Q} - \Delta hits(quantum\text{-}miss\text{-}time - quantum\text{-}hit\text{-}time)$$

wherein we remove/add the estimated excess cycles spent in serving the additional hits/misses respectively for the application with n ways. Hence, CAR_n is,

$$\frac{quantum\text{-}hits + quantum\text{-}misses}{Q - \Delta hits(quantum\text{-}miss\text{-}time - quantum\text{-}hit\text{-}time)}$$

It is important to note that extending ASM to estimate slowdowns for different cache allocations is straightforward since we use aggregate cache access rates. In contrast, extending previous slowdown estimation techniques such as FST and PTCA to estimate slowdowns for different cache allocations would require estimating *if every individual request would have been a hit/miss for every possible cache allocation*, which is non-trivial.

Cache Partitioning. Once we have each application's slowdown estimates for different way allocations, we use the look-ahead algorithm used in Utility-based Cache Partitioning (UCP) [56] to partition the cache ways such that the overall slowdown is minimized. Similar to the marginal miss utility (used by UCP), we define *marginal slowdown utility* as the decrease in slowdown per extra allocated way. Specifically, for an application with a current allocation of n ways, the marginal slowdown utility of allocating k additional ways is,

$$\text{Slowdown-Utility}_n^{n+k} = \frac{slowdown_n - slowdown_{n+k}}{k}$$

Starting from zero ways for each application, the marginal slowdown utility is computed for all possible way allocations for all applications. The application that has the maximum slowdown utility for a certain allocation is given those number of ways. This process is repeated until all ways are allocated. For more details on the partitioning algorithm, we refer the reader to [56].

7.1.2. Evaluation. Figure 9 compares the performance and fairness of ASM-Cache against a baseline that employs no cache partitioning (NoPart), Utility-based Cache Partitioning (UCP) [56] and memory-level parallelism and cache friendliness-aware quasi-partitioning scheme (MCFQ) [27] for different core counts. We simulate 100 workloads for each core count. We use harmonic speedup [19, 38] to measure system performance and the maximum slowdown metric [13, 30, 31, 61, 66, 69] (maximum slowdown in each workload and averaged over all workloads) to measure unfairness. Four observations are in order. First, ASM-Cache provides significantly better fairness and comparable/better performance across all core counts, over UCP. This is because ASM-Cache explicitly takes into account application slowdowns in performing cache allocation, whereas UCP uses miss counts as a proxy for performance. Second, MCFQ, while achieving fairness and performance benefits over UCP and comparable fairness and performance as ASM-Cache for workloads with low memory intensities, degrades fairness and performance for workloads with high memory intensities. This is because, although MCFQ takes into account memory-level parallelism and cache friendliness/interference, it does *not* take into account the impact of memory bandwidth interference. ASM-Cache, on the other hand, exploits slowdown estimates from ASM due to *both* main memory bandwidth and cache capacity interference for all possible way allocations. This enables it to estimate slowdown utility in a way that incorporates cache and memory interference. Third, ASM-Cache's gains increase with increasing core count: ASM-Cache reduces unfairness by 12.5% on the 8-core system and reduces unfairness by 15.8% and improves performance by 5.8% on the 16-core system, versus UCP. This is because contention for cache capacity increases with increasing core count, offering more opportunity for ASM-Cache to mitigate unfair slowdowns. Furthermore, the standard deviation of maximum slowdown for ASM-Cache across all our workloads is 3%/11%/13% lower than than of UCP, for the 4/8/16 core systems (not shown). Fourth, we see significant fairness improvements of 12.5% with a larger (4 MB) cache, on a 16-core system (plots not shown due to space constraints). We conclude that accurate slowdown estimates from ASM enables effective cache partitioning among contending applications, thereby improving both fairness and performance.

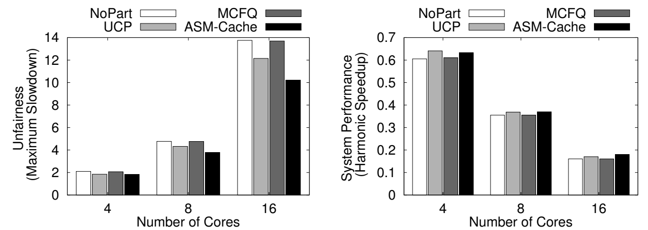

Figure 9: ASM-Cache: Fairness and performance

7.2. ASM Memory Bandwidth Partitioning

ASM-Mem partitions memory bandwidth among applications, based on slowdown estimates from ASM, with the goal of improving fairness. The basic idea behind ASM-Mem is to allocate bandwidth to each application proportional to its estimated slowdown, such that applications with higher slowdowns are given more bandwidth.

7.2.1. Mechanism. ASM is used to estimate all applications' slowdowns at the end of every quantum. These slowdown estimates are then used to determine the bandwidth allocation of each application. Specifically, the probability with which an epoch is assigned to an application is proportional to its estimated slowdown. The higher the slowdown of the application, the higher the probability that each epoch is assigned to the application. For an application A_i, the probability that an epoch is assigned to A_i is given by:

$$\text{Prob. of assigning an epoch to } A_i = \frac{slowdown(A_i)}{\sum_k slowdown(A_k)}$$

At the beginning of each epoch, the epoch is assigned to one of the applications based on the above probability distribution and requests of the corresponding application are prioritized over other requests during that epoch, at the memory controller.

7.2.2. Evaluation. We compare ASM-Mem with three previously-proposed memory schedulers, FRFCFS, PARBS and TCM. FRFCFS [58, 74] is an application-unaware scheduler that prioritizes row-buffer hits (to maximize bandwidth utilization) and older requests (for forward progress). FRFCFS tends to unfairly slow down applications with low row-buffer locality and low memory intensity [43, 46]. To tackle this problem, application-aware schedulers such as PARBS [47] and TCM [31] have been proposed. These reorder applications' requests at the memory controller, based on access characteristics.

Figure 10 shows the fairness and performance of ASM-Mem, FRFCFS, PARBS and TCM, for three core counts, averaged over 100 workloads for each core count. We draw four major observations. First, ASM-Mem achieves better fairness than the three previously-proposed schedulers, while achieving comparable/better performance. This is because ASM-Mem directly uses ASM's slowdown estimates to allocate more bandwidth to highly slowed-down applications, while previous works employ metrics such as memory intensity as proxies for performance/slowdown. Second, ASM-Mem's gains increase with core count, achieving 5.5% and 12% improvement in fairness on the 8- and 16-core systems respectively, compared to the fairest previous scheduler, PARBS. Third, ASM-Mem achieves fairness gains on systems with larger channel counts as well: 6% on a 16-core 2-channel system (not shown). Fourth, ASM-Mem's standard deviation of maximum slowdown across all our workloads reduces by 5%/7%/5% compared to the best previous mechanism, for the 4/8/16 core systems (not shown). We conclude that ASM-Mem is effective at mitigating interference at the main memory, thereby improving fairness.

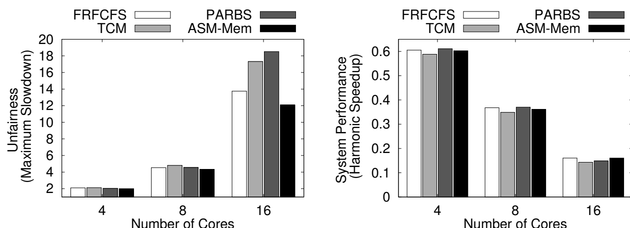

Figure 10: ASM-Mem: Fairness and performance

Combining ASM-Cache and ASM-Mem. We build a coordinated scheme, ASM-Cache-Mem, which performs cache partitioning using ASM-Cache and conveys the slowdown estimates for each application (corresponding to its cache way allocation) to the memory controller. The memory controller uses these slowdown estimates to partition memory bandwidth across applications using ASM-Mem. We compared ASM-Cache-Mem to combinations of previous cache partitioning and memory scheduling mechanisms, among which PARBS+UCP achieves both the best performance and fairness. ASM-Cache-Mem improves fairness by 14.6%/8.9% on a 16-core system with 1/2 channels, over PARBS+UCP, while achieving performance within 1% of PARBS+UCP.

7.3. Providing Soft Slowdown Guarantees

In multi-core systems where multiple applications are consolidated, ASM's slowdown estimates can be leveraged to *bound application slowdowns*. Figure 11 shows the slowdowns of four applications in a workload using a naive cache allocation scheme and a slowdown-aware scheme based on ASM. The goal is to achieve a specified slowdown bound for *h264ref*. The Naive-QoS scheme, which is unaware of application slowdowns, allocates all shared cache ways to *h264ref*, the application of interest. By doing so, it minimizes *h264ref*'s slowdown, thereby enabling it to meet any slowdown bound greater than 2.17. However, this comes at the cost of slowing down other applications significantly. ASM-QoS, on the other hand, allocates just enough cache ways to *h264ref* such that a specific slowdown bound (indicated by X in ASM-QoS-X) is met. Hence, ASM-QoS significantly reduces the other three applications' slowdowns compared to Naive-QoS.

This is an example policy that leverages ASM to partition the shared cache capacity to achieve a specific slowdown bound. We propose to explore more sophisticated schemes on top of ASM to control the allocation of memory bandwidth/cache capacity such that different applications' slowdown bounds are met, while still achieving high overall system performance.

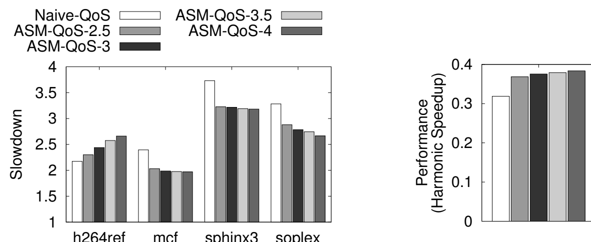

Figure 11: ASM-QoS: Slowdowns and performance

7.4. Fair Pricing in Cloud Systems

Applications from different users could be consolidated onto the same machine, e.g., in a server cluster. Pricing schemes in cloud systems bill users based on CPU core, memory/storage capacity allocation and run length of a job [1, 2]. ASM's slowdown estimates can enable taking into account shared resource interference. For instance, when jobs A and B are run together on the same system, job A runs for three hours due to cache/memory interference from job B, but would have run for only an hour, had it run alone. In this scenario, ASM would estimate job A's slowdown to be 3x, enabling the user to be billed for only one hour, versus three hours with a scheme that bills based only on resource allocation and run time.

7.5. Job Migration and Admission Control

ASM's slowdown estimates could be leveraged by the system software to make migration and admission control decisions. Previous works monitor different metrics such as cache misses and memory bandwidth utilization across machines/cores in a cluster or cores in a many-core system and migrate applications across machines based on these metrics [12, 36, 57, 67, 70]. While such metrics serve as proxies for interference, accurate slowdown estimates are a direct measure of the impact of interference on performance. Hence, periodically communicating slowdown estimates from ASM to the system software could enable better migration decisions. For instance, the system software could migrate applications away from machines on which slowdowns are very high or it could perform admission control and prevent new applications from being scheduled on machines where currently running applications are experiencing significant slowdowns, to avoid violating SLAs. Similarly, page migration mechanisms can potentially leverage ASM's slowdown estimates.

8. Related Work

We have already compared ASM to the closest previous works on estimating slowdowns due to shared cache and main memory interference, FST [15] and PTCA [14]. In this section, we compare to other related previous works.

Slowdown Estimation. Lin and Balasubramonian [33] propose a regression-based model to estimate performance for different cache allocations. Since this model does not take into account memory interference, it has a high error of 35% (across all our 4-core workloads).

Luque et al. [39] estimate slowdowns due to shared cache interference, but do not take into account main memory interference. Eyerman and Eeckhout [20] and Cazorla et al. [9] estimate slowdowns in SMT processors. ASM can be combined with these to estimate slowdowns due to both shared cache/memory interference and SMT.

Profiling. Prior works have attempted to quantify the impact of cache/memory contention through offline profiling. Mars et al. [40] estimate an application's sensitivity/propensity to receive/cause interference. Other previous works have proposed to estimate an application's sensitivity to cache capacity [16, 60] and memory bandwidth [17] through profiling. Yang et al. [72] attempt to estimate applications' sensitivity to interference online. However, this work assumes that latency-critical applications run alone at times, when they can be profiled (which could degrade system throughput).

The key distinction between these works and ASM is that these works assume the ability to profile applications offline or specific execution scenarios such as an application executing alone, while ASM can estimate the slowdown of any application *online*, in the general scenario of multiple applications running together.

Memory Interference Mitigation. Several previous works have tackled the problem of mitigating main memory interference with the goals of improving performance, fairness and quality of service. The major solution approach to mitigate memory interference has been application-aware memory scheduling [6, 21, 23, 25, 30, 31, 44, 46, 47, 49, 65]. ASM-Mem significantly improves fairness over state-of-the-art schedulers, while achieving comparable performance. Other prior approaches such as interleaving [28], channel/bank partitioning [26, 35, 45, 71], bandwidth partitioning [34, 66], source throttling [10, 15, 50, 51], thread scheduling [12, 67, 73] can be combined with ASM to further improve performance and fairness.

Cache Partitioning. Prior works have proposed cache partitioning schemes that achieve high performance and/or fairness [5, 27, 29, 56, 64]. ASM-Cache outperforms state-of-the-art cache partitioning techniques, UCP [56] and MCFQ [27]. ASM's slowdown estimates can be combined with metrics employed by other partitioning schemes to improve performance and fairness.

9. Conclusion

We introduce the Application Slowdown Model (ASM) to estimate the slowdowns of applications running concurrently on a multi-core system due to *both* shared cache and main memory interference. ASM accurately estimates slowdowns using the aggregate request behavior of each application. We demonstrate the effectiveness of ASM by using it to enable better shared resource management schemes to achieve different goals. We conclude that ASM is a promising substrate that can enable effective mechanisms to estimate and control application slowdowns in modern and future multi-core systems.

Acknowledgments

We thank the anonymous reviewers for their feedback. We acknowledge members of the SAFARI research group for their feedback. We acknowledge the generous support from our industrial partners: Google, Intel, Microsoft, Nvidia, Qualcomm, Samsung, VMware. This work is supported in part by NSF grants 0953246, 1212962, 1065112, 1320531, the Semiconductor Research Corporation, and the Intel Science and Technology Center on Cloud Computing. Lavanya Subramanian was a PhD student, Arnab Ghosh was an intern, and Samira Khan was a postdoctoral researcher, all in the SAFARI Group at CMU, when a majority of this work was carried out.

References

[1] *Amazon EC2 Pricing.* http://aws.amazon.com/ec2/pricing/.
[2] *Microsoft Azure Pricing.* http://azure.microsoft.com/en-us/pricing/details/virtual-machines/.
[3] *NAS Parallel Benchmark Suite.* http://www.nas.nasa.gov/publications/npb.html.
[4] *SPEC CPU2006.* http://www.spec.org/spec2006.
[5] K. Aisopos et al. Pcasa: Probabilistic control-adjusted selective allocation for shared caches. In *DATE*, 2012.
[6] R. Ausavarungnirun et al. Staged Memory Scheduling: Achieving high performance and scalability in heterogeneous systems. In *ISCA*, 2012.

[7] J.-L. Baer and T.-F. Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE TC*, May 1995.
[8] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *ACM Communications*, July 1970.
[9] F. Cazorla et al. Predictable Performance in SMT Processors: Synergy between the OS and SMTs. *IEEE TC*, 2006.
[10] K. Chang et al. HAT: Heterogeneous adaptive throttling for on-chip networks. In *SBAC-PAD*, 2012.
[11] B. Cooper et al. Benchmarking cloud serving systems with YCSB. In *SOCC*, 2010.
[12] R. Das et al. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *HPCA*, 2013.
[13] R. Das et al. Application-aware prioritization mechanisms for on-chip networks. In *MICRO*, 2009.
[14] K. Du Bois et al. Per-thread cycle accounting in multicore processors. In *HiPEAC*, 2013.
[15] E. Ebrahimi et al. Fairness via Source Throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *ASPLOS*, 2010.
[16] D. Eklov et al. Cache pirating: Measuring the curse of the shared cache. In *ICPP*, 2011.
[17] D. Eklov et al. Bandwidth bandit: Quantitative characterization of memory contention. In *PACT*, 2012.
[18] D. Eklov et al. A software based profiling method for obtaining speedup stacks on commodity multi-cores. In *ISPASS*, 2014.
[19] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, June 2008.
[20] S. Eyerman and L. Eeckhout. Per-thread cycle accounting in SMT processors. In *ASPLOS*, 2009.
[21] S. Ghose et al. Improving memory scheduling via processor-side load criticality information. In *ISCA*, 2013.
[22] A. Glew. MLP yes! ILP no! In *ASPLOS WACI*, 1998.
[23] E. Ipek et al. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.
[24] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *MICRO*, 2003.
[25] R. Iyer et al. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, 2007.
[26] M. K. Jeong et al. Balancing DRAM locality and parallelism in shared memory CMP systems. In *HPCA*, 2012.
[27] D. Kaseridis et al. Cache friendliness-aware managementof shared last-level caches for highperformance multi-core systems. *IEEE TC*, April 2014.
[28] D. Kaseridis et al. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *MICRO*, 2011.
[29] S. Kim et al. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.
[30] Y. Kim et al. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*, 2010.
[31] Y. Kim et al. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *MICRO*, 2010.
[32] Y. Kim et al. Ramulator: A fast and extensible DRAM simulator. *CAL*, 2015.
[33] X. Lin and R. Balasubramonian. Refining the utility metric for utility-based cache partitioning. In *WDDD*, 2009.
[34] F. Liu et al. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *HPCA*, 2010.
[35] L. Liu et al. A software memory partition approach for eliminating bank-level interference in multicore systems. In *PACT*, 2012.
[36] M. Liu and T. Li. Optimizing virtual machine consolidation performance on numa server architecture for cloud workloads. In *ISCA*, 2014.
[37] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.
[38] K. Luo et al. Balancing thoughput and fairness in SMT processors. In *ISPASS*, 2001.
[39] C. Luque et al. CPU accounting in CMP processors. *IEEE CAL*, January 2009.
[40] J. Mars et al. Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, 2011.
[41] Micron. Verilog: DDR3 SDRAM Verilog model.
[42] Micron. 2Gb DDR3 SDRAM, 2012.
[43] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security*, 2007.
[44] T. Moscibroda and O. Mutlu. Distributed order scheduling and its application to multi-core DRAM controllers. In *PODC*, 2008.
[45] S. P. Muralidhara et al. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *MICRO*, 2011.
[46] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO*, 2007.
[47] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA*, 2008.
[48] O. Mutlu et al. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA*, 2003.
[49] K. Nesbit et al. Fair queuing memory systems. In *MICRO*, 2006.
[50] G. Nychis et al. Next generation on-chip networks: What kind of congestion control do we need? In *HotNets*, 2010.
[51] G. Nychis et al. On-chip networks from a networking perspective: Congestion and scalability in many-core interconnects. In *SIGCOMM*, 2012.
[52] H. Patil et al. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO*, 2004.
[53] J. Pomerene et al. Prefetching system for a cache having a second directory for sequentially accessed blocks. Patent 407110 A, 1989.
[54] M. Qureshi et al. Adaptive insertion policies for high performance caching. In *ISCA*, 2007.
[55] M. Qureshi et al. A case for MLP-aware cache replacement. In *ISCA*, 2006.
[56] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.
[57] J. Rao et al. Optimizing virtual machine scheduling in numa multicore systems. In *HPCA*, 2013.
[58] S. Rixner et al. Memory access scheduling. In *ISCA*, 2000.
[59] P. Rosenfeld et al. DRAMSim2: A cycle accurate memory system simulator. *IEEE CAL*, January 2011.
[60] A. Sandberg et al. Modeling performance variation due to cache sharing. In *HPCA*, 2013.
[61] V. Seshadri et al. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *PACT*, 2012.
[62] T. Sherwood et al. Automatically characterizing large scale program behavior. In *ASPLOS*, 2002.
[63] S. Srinath et al. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.
[64] H. Stone et al. Optimal partitioning of cache memory. *IEEE TC*, September 1992.
[65] L. Subramanian et al. The blacklisting memory scheduler: Achieving high performance and fairness at low cost. In *ICCD*, 2014.
[66] L. Subramanian et al. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *HPCA*, 2013.
[67] L. Tang et al. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, 2011.
[68] Transaction Processing Performance Council. . http://www.tpc.org/.
[69] H. Vandierendonck and A. Seznec. Fairness metrics for multi-threaded processors. *IEEE CAL*, February 2011.
[70] H. Wang et al. A-DRM: Architecture-aware distributed resource management of virtualized clusters. In *VEE*, 2015.
[71] M. Xie et al. Improving system throughput and fairness simultaneously in shared memory CMP systems via dynamic bank partitioning. In *HPCA*, 2014.
[72] H. Yang et al. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ISCA*, 2013.
[73] S. Zhuravlev et al. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, 2010.
[74] W. Zuravleff and T. Robinson. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. Patent 5630096, 1997.