Prefetch-Aware Shared-Resource Management for Multi-Core Systems

Eiman Ebrahimi† *Chang Joo Lee*† *Onur Mutlu*‡ *Yale N. Patt*†

†High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

‡Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA

TR-HPS-2010-005
December 2010

This page is intentionally left blank.

Prefetch-Aware Shared-Resource Management for Multi-Core Systems

Eiman Ebrahimi† Chang Joo Lee† Onur Mutlu‡ Yale N. Patt†

†Department of ECE
Univ. of Texas at Austin
{ebrahimi, cjlee, patt}@ece.utexas.edu

‡Department of ECE
Carnegie Mellon Univ.
onur@cmu.edu

Abstract

Chip multiprocessor (CMP) systems share a large portion of the memory subsystem among multiple cores. Recent proposals have addressed high-performance and fair management of these shared resources; however, none of them take into account prefetch requests. Without prefetching, significant performance is lost, which is why existing systems prefetch. By not taking into account prefetch requests, all recent shared-resource management proposals often significantly degrade both performance and fairness, rather than improve them in the presence of prefetching.

This paper is the first to propose mechanisms that both manage the shared resources of a multi-core chip to obtain high-performance and fairness, and also exploit prefetching. We apply our proposed mechanisms to two resource-based management techniques for memory scheduling and one source-throttling-based management technique for the entire shared memory system. We show that our mechanisms improve the performance of a 4-core system that uses network fair queuing, parallelism-aware batch scheduling, and fairness via source throttling by 11.0%, 10.9%, and 11.3% respectively, while also significantly improving fairness.

1. Introduction

Chip multiprocessor (CMP) systems share a large portion of the memory subsystem among the multiple cores. This shared memory system typically consists of a last-level shared cache, on-chip interconnect, shared memory controllers and off-chip memory. When different applications concurrently execute on different cores of a CMP, they generate memory requests that interfere with memory requests of other applications in the shared memory resources. As a result of this inter-application interference, memory requests of different applications delay each another. This causes each application to slow down compared to when it runs in isolation. Recent research (e.g., [20, 19, 3]) has proposed different mechanisms to manage this interference in the shared resources in order to improve system performance and/or system fairness.

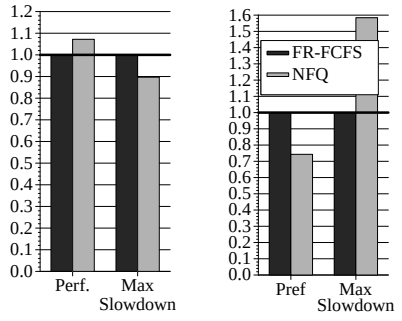
On the other hand, memory latency tolerance mechanisms are critical to improving system performance as DRAM speed continues to lag processor speed. Prefetching is one commonly-employed mechanism that predicts the memory addresses a program will require, and issues memory requests to those addresses before the program needs the data. Prefetching improves the standalone performance of many applications and is currently done in almost all commercial processors [30, 6, 11, 22]. Recent research [2] proposes intelligent dynamic adaptation of prefetcher aggressiveness to make prefetching effective and efficient in CMP systems.

Ideally we would like CMP systems to both obtain the performance benefits of prefetching when possible, and also reap the performance and fairness benefits of shared resource management techniques. However, shared resource management techniques that otherwise improve system performance and fairness significantly, can also significantly

degrade performance/fairness in the presence of prefetching. The reason: these techniques are designed for demand requests and do not consider prefetching.

Figure 1 illustrates this problem on a system that uses a fair/quality of service (QoS)-capable memory scheduler, network fair queuing (NFQ) scheduler [20]. Results are averaged over 15 multiprogrammed SPEC CPU2006 workloads on a 4-core system[1], and normalized to a system that uses a common first-ready first-come-first-serve (FR-FCFS) memory scheduler [26]. Figure 1 (a) shows how NFQ affects average system performance and average maximum slowdown (one metric of unfairness) in a system with no prefetching. Figure 1 (b) shows this in the presence of aggressive stream prefetching. This figure shows that, even though NFQ improves performance and reduces maximum slowdown on a system that does not have a prefetcher, if aggressive prefetching is enabled, we see a very different result. On a system with prefetching NFQ degrades performance by 25% while significantly increasing maximum slowdown, because its underlying prioritization algorithm does not differentiate between prefetch and demand requests. As a result, prefetches can be unduly prioritized by the memory scheduler, causing system performance and fairness degradation.

In this paper, we demonstrate that different shared resource management techniques suffer from this problem, i.e., they can degrade performance significantly when employed with prefetching. **Our goal** is to devise general mechanisms that intelligently take prefetches into account within shared resource management techniques to ensure their effectiveness for both performance and fairness in the presence of prefetching.



(a) No Prefetching (b) Stream Pref

Figure 1. Harmonic mean of speedups and maximum slowdown on system using NFQ memory scheduler (normalized to FR-FCFS)

We provide mechanisms for management of prefetch requests in three recently proposed shared resource management techniques. Two of these techniques are *resource-based* memory scheduling techniques: network fair queuing (NFQ) [20] and parallelism-aware batch scheduling (PARBS) [19]. The third technique is a *source throttling-based* technique for coordinated management of multiple shared resources (FST) [3].

Basic Ideas Our mechanisms build upon three fundamental ideas. First, we use accuracy feedback from the prefetchers to decide how prefetch requests should be handled in each of the *resource-based* techniques. The key idea is to *not* treat all prefetches the same. An application's prefetches should be treated similar to the demand requests *only when* they are useful.

Second, treating useful prefetches like demands can significantly delay demand requests of memory non-intensive

[1]Our system configuration, metrics, and workloads are discussed in Section 5. In Figure 1, the stream prefetcher of Table 1 is used. Prefetch and demand requests are treated alike with respect to NFQ's virtual finish time calculations.

applications because such requests can get stuck behind accurate prefetches (and demands) of memory-intensive applications. This degrades system performance and fairness. To solve this problem, we introduce the idea of *demand boosting*: the key idea is to boost the priority of the demand requests of memory non-intensive applications over requests of other applications.

Third, with *source throttling-based resource management*, we observe that uncoordinated core and prefetcher throttling can cause performance/fairness degradation because throttling decisions for cores can contradict those for prefetchers. To solve this problem, we propose mechanisms that coordinate core and prefetcher throttling based on interference feedback that indicates which cores are being unfairly slowed down.

Summary of Evaluation We evaluate our mechanisms on three different shared resource management techniques on a 4-core CMP system. Compared to a system with state-of-the-art prefetcher aggressiveness control [2], we find that our mechanisms improve the performance of an NFQ-based, PARBS-based, and FST-based system on average by 11.0%, 10.9%, and 11.3% while at the same time reducing maximum slowdown by 9.9%, 18.4%, and 14.5%.

Contributions This paper makes the following contributions:

1. It demonstrates a new problem in multi-core shared resource management: prefetching can significantly degrade system performance and fairness of multiple state-of-the-art shared resource management techniques. This problem still largely exists even if state-of-the-art prefetcher throttling techniques are used to dynamically adapt prefetcher aggressiveness.

2. It shows that simply prioritizing accurate prefetches and deprioritizing inaccurate ones within shared resource management techniques does not solve the problem; prioritized prefetches can significantly degrade the performance of memory non-intensive applications. We introduce the idea of demand boosting to prevent this.

3. It introduces new general mechanisms to handle prefetches in shared resource management techniques to synergistically obtain the benefits of both prefetching and shared resource management techniques in multi-core systems. We apply our mechanisms to three state-of-the-art shared resource management techniques and demonstrate in detail how these techniques should be made aware of prefetching. Comprehensive experimental evaluations show that our proposal significantly improves fairness and performance of these techniques in the presence of prefetching.

2. Background

In the sections that follow, we briefly describe the three different shared resource management techniques that we discuss in this paper. We also give a brief introduction to a state-of-the-art prefetcher control technique [2] that improves system performance and fairness in the presence of prefetching in CMP systems. We first briefly describe what we mean by system fairness.

2.1. Fairness in the Presence of Prefetching

We evaluate fairness of a multi-core system executing a multi-programmed workload using the $MaxSlowdown$ metric. This metric shows the maximum individual slowdown that any application in the workload experiences, as an indicator of the minimum service that any application in the workload receives. *Individual Slowdown (IS)* of each application is calculated as T_{shared}/T_{alone}, where T_{shared} is the number of cycles it takes an application to run simultaneously with other applications, and T_{alone} is the number of cycles it would have taken the application to run alone on the same system. In all of our evaluations, we use an aggressive stream prefetcher when calculating each benchmark's T_{alone} as our stream prefetcher significantly improves average performance and makes for a better baseline system. In addition to the $MaxSlowdown$ metric, we also show the commonly used *unfairness* metric [9, 4, 18] calculated as:

$$Unfairness = \frac{MAX\{IS_0, IS_1, ..., IS_{N-1}\}}{MIN\{IS_0, IS_1, ..., IS_{N-1}\}}$$

2.2. Network Fair Queuing Memory Scheduling

Nesbit et al. [20] propose network fair queuing (NFQ), a memory scheduling technique based on the concepts of fair network scheduling algorithms. NFQ's goal is to provide quality of service to different concurrently executing applications based on each application's assigned fraction of memory system bandwidth. NFQ's QoS objective is that "a thread *i* that is allocated a fraction *F* of the memory system bandwidth will run no slower than the same thread on a private memory system running at that fraction *F* of the frequency of the shared physical memory system." NFQ determines a *virtual finish time* for every request of each thread. A memory request's virtual finish time is the time it would finish on the thread's virtual private memory system (a memory system running at the fraction *F* of the frequency of the shared memory system). To achieve this objective, memory requests are scheduled *earliest virtual finish time first*. NFQ provides no specification of how prefetches should be treated.

2.3. Parallelism-Aware Batch Scheduling

Mutlu and Moscibroda [19] propose parallelism-aware batch scheduling (PARBS), a memory scheduling technique aimed at improving throughput by preserving intra-thread bank parallelism while providing fairness by avoiding starvation of requests from different threads.[2] There are two major steps to the PARBS algorithm: First, PARBS generates batches from a number of outstanding memory requests, and ensures that all requests belonging to the current batch are serviced before the formation of the next batch. This batching technique avoids starvation of different threads and is aimed at improving system fairness. Second, PARBS preserves intra-thread bank-level-parallelism while servicing requests from each application within a batch. This step improves system throughput by reducing each thread's

[2]We assume each core of a CMP runs a separate application, and use the term thread and application interchangeably.

memory related stall time. PARBS does not specify how to handle prefetches in either of these two steps.

2.4. Hierarchical Prefetcher Aggressiveness Control

Ebrahimi et al. [2] propose hierarchical prefetcher aggressiveness control (HPAC) as a prefetcher throttling solution to improve prefetching performance in CMPs. HPAC's goal is to control/reduce inter-thread interference caused by prefetchers. It does so by gathering global feedback information about the effect of each core's prefetcher on concurrently executing applications. Examples of global feedback are memory bandwidth consumption of each core, how much each core is delayed waiting for other applications to be serviced by DRAM, and cache pollution caused by each core's prefetcher for other applications in the shared cache. Using this feedback, HPAC throttles each core's prefetcher. By doing so, Ebrahimi et al. [2] show that HPAC can enable system performance improvements using prefetching that are not possible without it. In our paper, we use HPAC in our baseline system since it significantly improves the performance of prefetching in multi-core systems and therefore constitutes a stronger baseline.

2.5. Fairness via Source Throttling

Ebrahimi et. al. [3] propose fairness via source throttling (FST) as an approach to providing fairness in the entire shared memory system. FST dynamically estimates how much each application *i* is slowed down due to inter-core interference that results from sharing the memory system with other applications. Using these estimated slowdowns, FST calculates an estimate for system unfairness. In addition, FST also determines the core experiencing the largest slowdown in the system, referred to as *App-slowest*, and the core creating the most interference for *App-slowest*, referred to as *App-interfering*. If the estimated unfairness is greater than a threshold specified by system software, FST throttles down *App-interfering* (i.e., it reduces how aggressively that application accesses the shared memory resources), and throttles up *App-slowest*. In order to throttle down the interfering thread, FST limits the number of requests that the thread can simultaneously send to the shared resources and also the frequency at which it does so.

In order to estimate each application's slowdown, FST tracks inter-thread interference in the memory system. FST estimates *both* how much each application *i* is actually being slowed down due to inter-core interference and *also* how much each other core *j* ($j \neq i$) contributes to the interference experienced by core *i*. Unfortunately, FST assumes all requests are demand requests and does not consider prefetching.

3. Motivation

In this section, we motivate why special treatment of prefetch requests is required in shared resource management techniques to both 1) achieve benefits from prefetching and, 2) maintain the corresponding techniques' performance benefits and/or fairness/QoS capabilities.

Every shared resource management technique has a prioritization algorithm that determines the order in which

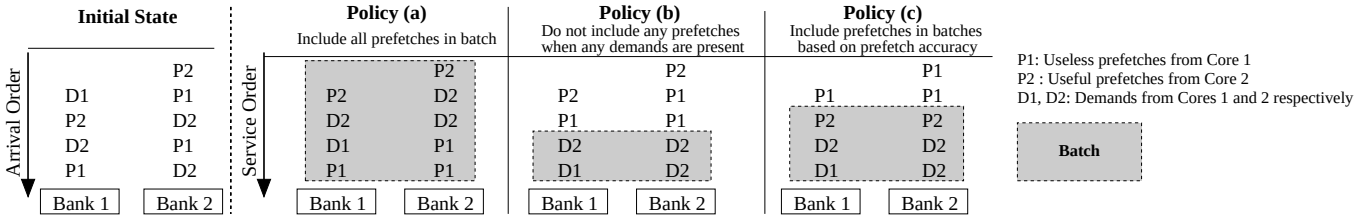


Figure 2. Example 1: Different policies for treatment of prefetches in PARBS batch formation

requests are serviced. For example, NFQ prioritizes service to requests that have earlier *virtual finish times*. PARBS prioritizes requests included in the formed batch by scheduling them all before a new batch is formed. In resource-based management techniques, the first key idea of our proposal is that usefulness of prefetch requests should be considered within each management technique's prioritization policy. As such, not all prefetches should be treated the same as demand requests, and not all prefetches should be deprioritized compared to demand requests. However, this is not enough; in fact, prioritizing accurate prefetches causes starvation to demands of non-intensive applications. To solve this problem, the second key idea of our proposal is to boost the priority of demand requests of such non-intensive applications so that they are not starved.

We motivate these two key ideas with two examples.

Example 1: Figure 2 shows the effect of prefetching on PARBS. The figure shows a snapshot of the memory request buffers in the memory controller for banks 1 and 2. The initial state of these queues right before a new batch is formed can be seen on the left. Based on PARBS's batching algorithm, a maximum number of requests from any given thread to any given bank are marked to form a batch. Let us assume PARBS marks three requests per-thread per-bank when forming a batch. Additionally, let us assume that application 1's prefetches are useless or inaccurate while application 2's prefetches are useful or accurate. Figure 2 shows two simplistic policies, (a) and (b), and our proposed approach, policy (c), for handling prefetches in PARBS's batching phase. Figure 3 shows the respective memory service timelines.

Policy (a): mark prefetches and demands from each thread alike when creating a batch. Figure 2 shows that all the requests in the memory request queues of the two banks are included in the batch with this policy. Within each batch, PARBS prioritizes threads that that are "shorter jobs" in terms of memory request queue length. Since thread 1 has a shorter queue length (maximum 2 requests in any bank) than thread 2 (maximum 3 requests in any bank), thread 1 is prioritized over thread 2. As a result, as Figure 3 (a) shows, thread 1's inaccurate prefetches to addresses Y, X and Z are prioritized over thread 2's demands and useful prefetches. This leads to unwarranted degradation of thread 2's performance without any benefit to thread 1 (as its prefetches are useless).

Policy (b): never mark prefetches. This policy provides a naive solution to policy (a)'s problems by not marking any

prefetches. This is helpful in prioritizing the demands of thread 2 over the useless prefetches of thread 1. However, by not marking any prefetches, this policy also does not include the useful prefetches of thread 2 in the generated batch. Figure 3 (b) shows that thread 2's useful prefetches to addresses L and M are now delayed since all prefetches are deprioritized. Hence thread 2 issues demands for addresses L and M before the prefetches are serviced, and so the benefit of those accurate prefetches significantly decreases. This causes a loss of potential performance.

Our Approach: A key principle in this paper is to treat only accurate prefetches as demands in shared resource management. Figure 2 (c) concisely shows how this is done for PARBS. Using feedback from different threads' prefetchers, PARBS can make a more intelligent decision about whether or not to include prefetches when forming batches. Since thread 2's prefetches are useful, we include them in the batch, while thread 1's useless prefetches are excluded. As a result, benefits from prefetching for thread 2 is maintained, as shown in Figure 3 (c). Excluding thread 1's useless prefetches from the batch improves system fairness as these requests do not unduly delay thread 2's demands and useful prefetches, and thread 2's slowdown is reduced without increasing thread 1's slowdown. Figure 3 (c) shows that this policy improves both applications' performance compared to policies that treat all prefetches equally, motivating the need for distinguishing between accurate and inaccurate prefetches in shared resource management.

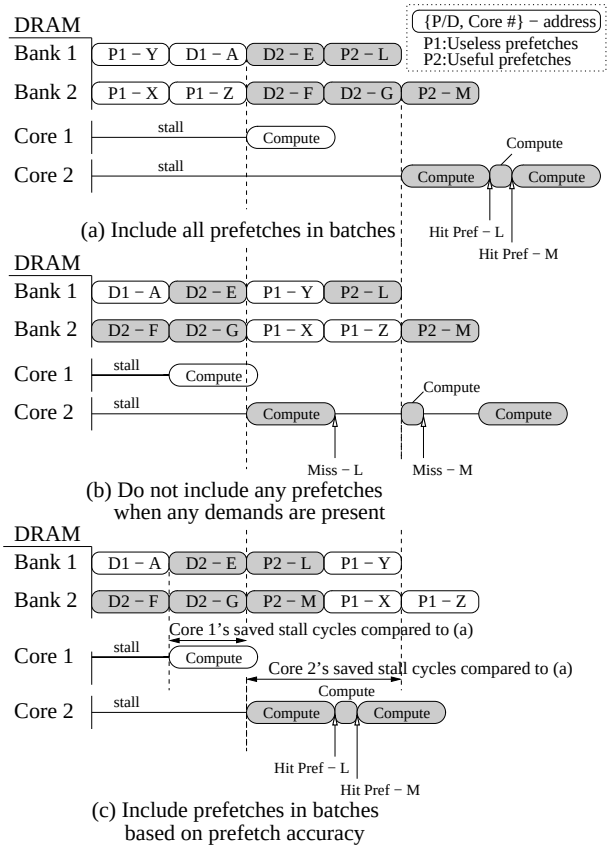


Figure 3. Memory service timeline for requests of Figure 2

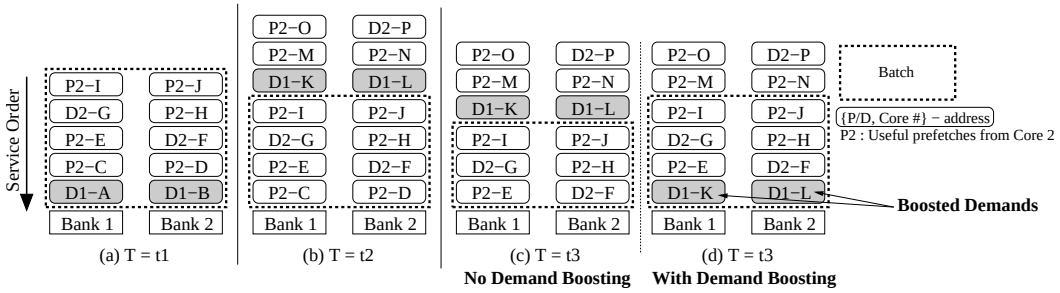


Figure 4. Example 2: No demand boosting vs. Demand boosting

Example 2: Figure 4 shows the problem with just prioritizing accurate prefetches, and concisely shows our solution for a system using PARBS. When including accurate prefetches into the batches formed by PARBS, in the presence of prefetch-friendly applications (like application 2 in Figure 4), the size of the batches can increase. Since memory non-intensive applications (like application 1 in Figure 4) generate memory requests at a slow pace, every time a batch is formed (Time *t1* shown in Figure 4(a)), memory non-intensive applications will have a small number of their requests included. At time *t2*, more requests from the memory non-intensive application arrive. Without our proposed mechanism, since the current batch is still being serviced, these requests have to wait until the current batch is finished (Figure 4 (c)), which could take a long time since useful prefetch requests that were entered into the batch made the batch size larger. In this paper, we propose demand boosting, which prioritizes the *small number* of the non-intensive application's requests over others. In Figure 4 (d), at time *t3*, the two demand requests from application 1 to addresses K and L are boosted *into* the current batch and prioritized over the existing requests from application 2 within the batch. This allows application 1 to go back to its compute phase quickly. Doing so does not degrade application 2's performance significantly as the non-intensive application 2 inherently has very few requests.

4. High Performance and Fair Shared Resource Management in the Presence of Prefetching

In this section, we describe in detail our proposal for handling prefetches in the two types of resource management techniques: *resource-based* and *source-based*. We also introduce *demand boosting*, which is orthogonal to the employed resource management technique. Since demand boosting is common to both resource-based and source-based techniques, we describe it first in Section 4.1. Then, we describe in detail how to apply our insights (described in Sections 1 and 3) to each resource management technique in turn (Sections 4.2 and 4.3).

4.1. Demand Boosting

Problem and Main Idea: As described in Section 3, the first component of our proposal is to treat useful prefetches to be as important as the demands. Memory-intensive and prefetch-friendly applications can generate many such requests, which can cause long delays for the demands of concurrently executing non-intensive threads. As a result, system performance and fairness can degrade because of large performance degradations to memory non-intensive applications. To mitigate this problem, we propose *demand boosting* for such non-intensive applications. The key idea is to prioritize the non-intensive application's *small number of demand requests* over others, allowing that application to go back to its compute phase quickly. It must be noted that doing so does not significantly degrade other applications' performance because the non-intensive application inherently has very few requests.

Why the Problem Exists: The potential for *short-term* starvation of a non-intensive application's demands increases in each of the techniques we consider for different reasons. In NFQ and FST, potential for starvation is created

by the prioritization of DRAM row buffer hits in the memory scheduler, coupled with high row buffer locality of accurate prefetches that are considered as important as demands. PARBS uses the batching concept to mitigate this inherent issue due to prioritizing row-buffer hit requests. However, in Section 3 we proposed including accurate prefetches into PARBS's batches. The slow rate at which non-intensive threads generate their requests, together with the large batches generated using requests from prefetch-friendly applications, causes potential for starvation in PARBS.

To summarize, elevating the priority of accurate prefetch requests from memory intensive applications causes the small memory related stall times of non-intensive applications to increase. This significantly hurts the non-intensive applications' performance (as also observed by prior work [12]). In addition, when such memory non-intensive applications are cache friendly, as they stall waiting for their small number of memory requests to be serviced, their useful requests in the shared cache move up the LRU stack and can get evicted more quickly by intensive applications' requests. This, in turn, causes larger performance penalties for such memory non-intensive applications.

Demand Boosting Mechanism: Demand boosting is a general mechanism orthogonal to the type of resource management technique. It increases the performance of memory non-intensive applications that do not take advantage of accurate prefetches by dynamically prioritizing *a small number* of such applications' demands. With demand boosting, the demands of an application that does not have accurate prefetches ***and*** has a at most a *threshold number* of outstanding requests, will be boosted and prioritized over *all other* requests. For example, in a system using PARBS, when an application's demands are boosted, they no longer wait for a current batch to finish before they are considered for scheduling. A boosted request X has higher priority than any other request Y regardless of whether or not request Y is in the current batch.[3]

Delaying a memory-intensive application in lieu of a memory non-intensive application with inherently small memory stall times can improve both system performance and fairness [19, 13, 3, 10]. In many cases, demand-boosting enables performance benefits from prefetching that are not possible without it, as we show in Section 6.

4.2. Prefetch-Aware Resource-Based Management Techniques

We identify prefetcher accuracy as the critical prefetcher characteristic to determine how a prefetcher's requests should be treated in shared resource management techniques. Prefetcher accuracy is defined as the ratio of useful prefetches generated by a prefetcher to the total number of prefetches it generates. We also investigated using other prefetcher feedback such as a prefetcher's *degree of timeliness*[4], but found that accuracy has more of a first order

[3]Note that in the context of demand boosting for PARBS, demand boosting is significantly different from the "intra-batch" ranking proposed by the original PARBS mechanism (which we use in all our PARBS related mechanisms). PARBS's ranking prioritizes requests chosen from requests already contained *within the current batch* using its ranking algorithm. In contrast, with demand boosting, demand requests from a boosted thread are prioritized over *all* other requests.

[4]A prefetcher's degree of timeliness is defined as the ratio of the number of useful prefetches that fill the last level cache before the corresponding demand request is issued, to the total number of useful prefetches.

effect.

In all of the mechanisms we propose, we measure prefetch accuracy on an interval by interval basis. An interval ends when $T = 8192$ cache lines are evicted from the last level cache, where T is empirically determined. Every interval, feedback information on the number of useful prefetches and total sent prefetches of each prefetcher is gathered. Using this feedback information, the accuracy of the prefetcher in that interval is calculated and used as an estimate of the prefetcher accuracy in the following interval. In the following subsections, we discuss how to redesign underlying prioritization principles of the different techniques.

4.2.1. Parallelism-Aware Batch Scheduling PARBS uses *batching* to provide a minimum amount of DRAM service to each application by limiting the maximum number of requests considered for scheduling from any one application. Inaccurate prefetches of an application A can have negative impact on system performance and fairness in two ways. First, they get included in batches and get prioritized over other applications' demands and useful prefetches that were not included. As a result, they cause large performance degradation for those other applications without improving application A's performance. Second, they reduce the fairness provided by PARBS to application A by occupying a number of slots of each batch that would otherwise be used to give application A's demands a minimum amount of useful DRAM service.

We propose the following new batch scheduling algorithm to enable potential performance improvements from prefetching, while maintaining the benefits of PARBS. The key to Algorithm 1 is that it restricts the process of marking requests to demands and accurate prefetches. As a result, a prefetch-friendly application will be able to benefit from prefetching within its share of memory service. On the other hand, inaccurate requests are not marked and are hence deprioritized by PARBS.

Algorithm 1 Parallelism-Aware Batch Scheduler's Batch Formation (Prefetch-Aware PARBS, P-PARBS)

Forming a new batch: A new batch is formed when there are no marked requests left in the memory request buffer, i.e., when all requests from the previous batch have been completely serviced.
Marking: When forming a new batch, the scheduler marks up to *Marking-Cap* outstanding demand ***and also accurate prefetch requests*** for each application; these requests form the new batch.

4.2.2. Network Fair Queuing NFQ uses *earliest virtual finish time first* memory scheduling to provide quality of service to concurrently executing applications. Inaccurate prefetches of some application A can have negative impact on system performance and fairness in two ways: First, if application A's inaccurate prefetches get prioritized over demands or accurate prefetches of some other application B due to the former's earlier virtual finish time, system performance will degrade. Application B's service is delayed while application A does not gain any performance. Second, since NFQ provides service to application A's inaccurate prefetches, the virtual finish times of application

A's demands grows larger than when there was no prefetching. This means that application A's demand requests will get serviced later compared to when there is no prefetching. Since application A's prefetches are not improving its performance, this ultimately results in application A's performance loss due to unwarranted waste of its share of main memory bandwidth.

We propose the following prioritization policy for the NFQ bank scheduler. When this scheduler prioritizes requests based on earliest virtual finish time, this prioritization is performed only for demand accesses and *accurate* prefetches. Doing so prevents the two problems described in the previous paragraph. Algorithm 2 summarizes the proposed NFQ policy.

Algorithm 2 Network Fair Queuing's Bank Scheduler Priority Policy (Prefetch-Aware NFQ, P-NFQ)

Prioritize ready commands (highest)
Prioritize CAS commands
Prioritize commands for demands ***and also accurate prefetch requests*** with earliest virtual finish-time
Prioritize commands based on arrival time (lowest)

4.3. Prefetch-Aware Source-Based Management Techniques

We propose prefetch handling mechanisms for a recent *source-based* shared resource management approach, FST [3]. We briefly described FST's operation in Section 2.5. FST does not take into account *interference generated for prefetches* and *interference generated by the prefetches* of each application.

We incorporate prefetch awareness into FST in two major ways by: a) determining how prefetches and demands should be considered in estimating slowdown values, and b) coordinating core and prefetcher throttling using FST's monitoring mechanisms.

4.3.1. Determining Application Slowdown in the Presence of Prefetching FST tracks interference in the shared memory system to dynamically estimate the slowdown experienced by each application. Yet, it cannot compute accurate slowdown values if prefetching is employed because FST is unaware of prefetches. We describe a new mechanism to compute slowdown when prefetching is employed.

When requests A and B from two applications interfere with each other in a shared resource, one request receives service first and the other is *interfered-with*. Let us assume that request A was the *interfering* and request B was the *interfered-with*. The *type* of memory request A classifies the interference as *prefetch-caused* or *demand-caused* interference. The type of memory request B classifies the interference as *prefetch-delaying* or *demand-delaying* interference.

FST defines individual slowdown, *IS*, as T_{shared}/T_{alone} to estimate system unfairness. In order to estimate T_{alone} when running in shared mode, FST makes an estimation of "the number of *extra cycles* it takes an application to

execute due to inter-core interference in the shared memory resources." This is known as T_{excess} ($T_{excess} = T_{shared} - T_{alone}$).

When estimating T_{excess} in the presence of prefetching, we find that it is important to use the following two principles. First, both *prefetch-caused* and *demand-caused* interference should be considered. Second, only *demand-delaying* interference should be used to calculate slowdown values at runtime. This means that when calculating core *i*'s T_{excess}, interference caused for its demands by *either* demands or prefetches of other cores *j* ($j \neq i$) should be accounted for. This is because ultimately both prefetch and demand requests from an interfering core can cause an *interfered-with* core to stall. On the other hand, even though *prefetch-delaying* interference reduces the timeliness of interfered-with prefetches, it does not significantly slow down the corresponding core. If an accurate prefetch is delayed until the corresponding demand is issued, that prefetch will be promoted to a demand. Further delaying of that request will contribute to the slowdown estimated for the respective core because any interference with that request will be considered *demand-delaying* from that point on.

Algorithm 3 summarizes how our proposal handles prefetches to make FST prefetch-aware.[5] FST uses a bit per core to keep track of when each core was interfered with. We refer to this bit-vector as the *Interference* bit-vector in the algorithm. Also, an $ExcessCycles$ counter is simply used to track T_{excess} for each core.

Algorithm 3 Prefetch-aware FST (P-FST) estimation of T_{excess} for core *i*

Every cycle
if inter-core interference created by any core *j*'s ***prefetch requests*** or ***demand requests*** for core *i*'s ***demand requests*** **then**
  set core *i*'s bit in the $Interference$ bit-vector
end if
if Core *i*'s bit is set in the $Interference$ bit-vector **then**
  Increment $ExcessCycles$ counter for core *i*
end if

4.3.2. Coordinated Core and Prefetcher Throttling FST throttles cores to improve fairness and system performance. On the other hand, HPAC is an independent technique that throttles prefetchers to improve system performance by controlling prefetcher-caused inter-core interference. Unfortunately, combining them without coordination causes contradictory decisions. For example, the most slowed down core's prefetcher can be throttled down (by the prefetch throttling engine, i.e., HPAC's global control) while the core is being throttled up (by the core throttling engine, i.e. FST). As a result, fairness and performance degrade and potential performance benefits from prefetching can be lost. Therefore, we would like to coordinates the decisions of core and prefetcher throttling. The key insight is to inform/coordinate HPAC's throttling decisions with FST's decisions using the interference information collected by FST. We achieve this in two ways.

[5] We present our changes to the original T_{excess} estimation algorithm [3]. For other details on T_{excess} estimation we refer to [3].

The first key idea is to use the slowdown information that FST gathers for core throttling to make better prefetcher throttling decisions. To do this, we only apply HPAC's global prefetcher throttle down decisions to a core if FST has detected the corresponding core to be $App_{interfering}$. [6] As such, we *filter* some of the throttle-down decisions made by HPAC. This is because HPAC can be very strict at prefetcher throttling due to its coarse classification of the severity of prefetcher-caused interference. As a result, it throttles some prefetchers down *conservatively* even though they are not affecting system performance/fairness adversely. We avoid this by using the information FST gathers about which cores are actually being treated unfairly as a result of inter-core interference.

The second key idea is to use FST's ability in tracking inter-core cache pollution to improve how well HPAC detects accurate prefetchers. This is useful because HPAC can underestimate a prefetcher's accuracy due to its interference-unaware tracking of useful prefetches. HPAC does not count accurate prefetches for core *i* that were evicted by some other core's requests before being used. This can cause HPAC to incorrectly throttle down core *i*'s accurate prefetcher and degrade its performance. To avoid this, we use FST's pollution filter to detect when an accurate prefetch for core *i* was evicted due to another core *j*'s request. For this purpose, we extend FST's pollution filter entries to also include a prefetch bit. Using this, we account for useful prefetches evicted by another core's requests in HPAC's estimation of each prefetcher's accuracy.

Algorithms 4 and 5 summarize the above mechanisms that coordinate core and prefetcher throttling.

Algorithm 4 Prefetch-Aware FST (P-FST) Core and Prefetcher Throttling

if $Estimated\,Unfairness > Unfairness\,Threshold$ **then**
  Throttle down $App_{interfering}$
  Throttle down prefetcher of $App_{interfering}$ **if HPAC indicates global throttle down for this prefetcher**
  Throttle up $App_{slowest}$
end if
Allow HPAC to throttle up prefetchers as it requires
Apply HPAC's local throttle down decisions

Algorithm 5 Enhancing prefetcher accuracy information using FST's pollution filters

if Last-level cache hit on prefetched cache line **then**
  increment useful prefetch count
end if
if Last-level cache miss due to inter-core interference as detected by FST **and** evicted line was prefetch request **then**
  increment useful prefetch count
end if
Prefetch accuracy = useful prefetch count / total prefetch count

5. Methodology

Processor Model: We use an in-house cycle-accurate x86 CMP simulator for our evaluation. We faithfully model

[6] If HPAC's local throttling component fore core *i* detects that the core's prefetcher is not performing well, that prefetcher is still throttled down regardless of FST's decision. This helps both core *i*'s and other cores' performance.

all port contention, queuing effects, bank conflicts, and other major DDR3 DRAM system constraints in the memory system. Table 1 shows the baseline configuration of each core and the shared resource configuration for the 4-core CMP system we use.

Execution core	15 stage out of order processor, decode/retire up to 4 instructions Issue/execute up to 8 micro instructions; 128-entry reorder buffer
Front end	Fetch up to 2 branches; 4K-entry BTB; 64K-entry Hybrid branch predictor
On-chip caches	L1 I-cache: 32KB, 4-way, 2-cycle, 64B line ; L1 D-cache: 32KB, 4-way, 2-cycle, 64B line Shared unified L2: 2MB , 16-way, 16-bank, 20-cycle, 1 port, 64B line size
Prefetcher	Stream prefetcher with 32 streams, prefetch degree of 4, and prefetch distance of 64 cache lines [30, 28]
DRAM controller	On-chip, Open-row PARBS [19]/NFQ [20]/FR-FCFS [26] scheduling policies 128-entry MSHR and memory request queue
DRAM and bus	667MHz bus cycle, DDR3 1333MHz [17] 8B-wide data bus, 8 DRAM banks, 16KB row buffer per bank Latency: 15-15-15ns (^tRP-^tRCD-CL), corresponds to 100-100-100 processor cycles Round-trip L2 miss latency: Row-buffer hit: 36ns, conflict: 66ns

Table 1. Baseline system configuration

Benchmarks: We use the SPEC CPU 2000/2006 benchmarks for our evaluation. Each benchmark was compiled using ICC (Intel C Compiler) or IFORT (Intel Fortran Compiler) with the -O3 option. Each benchmark runs the reference input set for 50 million x86 instructions selected by Pinpoints [23].

We classify a benchmark as *memory-intensive* if its L2 Cache Misses per 1K Instructions (MPKI) is greater than three and otherwise we refer to it as *memory non-intensive.* We say a benchmark has *cache locality* if the number of L2 cache hits per 1K instructions for the benchmark is greater than five. An application is classified as *prefetch-friendly* if its IPC improvement due to prefetching when run in isolation is more than 10%. If its IPC degrades, it is classified as *prefetch-unfriendly* and otherwise as *prefetch-insensitive.* These classifications are based on measurements made when each benchmark was run alone on the 4-core system. Table 2 shows the characteristics of 18 of the 29 benchmarks (due to space limitations) that appear in the evaluated workloads when run on the 4-core system.

Workload Selection We used 15 four-application workloads for our evaluations. The workloads were chosen such that each workload consists of at least two *memory-intensive* applications (MPKI greater than three) and an application *with cache locality*. All but one workload has at least one *prefetch-friendly* application since the goal of the paper is to demonstrate how to improve system performance due to prefetching in systems that employ the different shared resource management mechanisms. The one workload with no prefetch-friendly applications consists of memory-intensive and prefetch-unfriendly applications.

Parameters used in evaluation: In all our mechanisms, the threshold to determine whether an application's prefetcher is accurate is 80%. In P-NFQ and P-FST, an application must have *fewer than* ten memory requests in the memory request queue of the memory controller to be considered for *demand boosting,* and fewer than 14 requests in P-PARBS (Section 6.5 shows that the reported results are not very sensitive to the value chosen for this threshold).

		No prefetching		Prefetching							No prefetching		Prefetching				
Benchmark	Type	IPC	MPKI	IPC	MPKI	HPKI	Acc(%)	Cov(%)	Benchmark	Type	IPC	MPKI	IPC	MPKI	HPKI	Acc(%)	Cov(%)
art	FP00	0.23	25.7	0.25	13.73	105	61	55	sphinx3	FP06	0.26	12.82	0.51	2.71	14.5	58	79
gromacs	FP06	1.17	0.22	1.2	0.07	11	66	70	leslie3d	FP06	0.29	21.37	0.55	4.73	22.3	94	78
lbm	FP06	0.33	19.3	0.36	3.43	27.4	94	82	bwaves	FP06	0.26	22.43	0.33	2.3	11.3	100	90
GemsFDTD	FP06	0.38	12.67	0.67	0.07	17.6	93	99	astar	INT06	0.17	23.04	0.17	21.4	10.4	25	8
omnetpp	INT06	0.34	8.79	0.34	8.72	5	11	19	vortex	INT00	0.97	1.21	0.93	1.15	7	27	14
zeusmp	FP06	0.66	3.97	0.75	1.92	17	67	52	swim	FP00	0.39	16.85	0.48	0.57	20	100	97
bzip2	INT06	1.57	0.96	1.65	0.64	7.8	95	35	h264ref	INT06	1.89	0.77	1.86	0.43	2	56	55
perlbmk	INT00	1.8	0.04	1.8	0.03	5.4	16	35	crafty	INT00	1.56	0.26	1.61	0.19	8	34	29
xalancbmk	INT06	1.07	0.83	0.93	0.99	18.8	11	18	libquantum	INT06	0.26	11.84	0.29	2.21	0.52	100	81

Table 2. Characteristics of 18 SPEC 2000/2006 benchmarks: IPC and MPKI (L2 cache Misses Per 1K Instructions) with and without prefetching, HPKI (L2 cache Hits Per 1K Instructions) with prefetching, and prefetcher accuracy and coverage

The parameter setup for each of the FST and HPAC techniques is the same as those reported in [3] and [2] respectively. For PARBS [19], we use the same *Marking Cap* threshold as used in the original paper, five memory requests per thread per bank.

Metrics: To measure CMP system performance, we use *Harmonic mean of speedups (Hspeedup)* [16], and *Weighted speedup (Wspeedup)* [27]. To demonstrate fairness improvements, we report *MaxSlowdown*, and also *Unfairness* as defined in [4, 18] (see Section 2.1). Since *Hspeedup* provides a balanced measure between fairness and system throughput [16], we use it as our primary evaluation metric. In the metric definitions below: N is the number of cores in the CMP system, IPC^{alone} is the IPC measured when an application runs alone on one core in the CMP system (other cores are idle), and IPC^{shared} is the IPC measured when an application runs on one core while other applications are running on the other cores.

$$Hspeedup = \frac{N}{\sum_{i=0}^{N-1} \frac{IPC_i^{alone}}{IPC_i^{shared}}}, \quad Wspeedup = \sum_{i=0}^{N-1} \frac{IPC_i^{shared}}{IPC_i^{alone}}$$

6. Experimental Evaluation

We evaluate the mechanisms described in the previous sections on a 4-core CMP system employing NFQ, PARBS, and FST in the following three subsections respectively. Note that our prefetch-aware NFQ, PARBS, and FST techniques (P-NFQ, P-PARBS, and P-FST) are evaluated on a system with state-of-the-art prefetcher throttling [2].

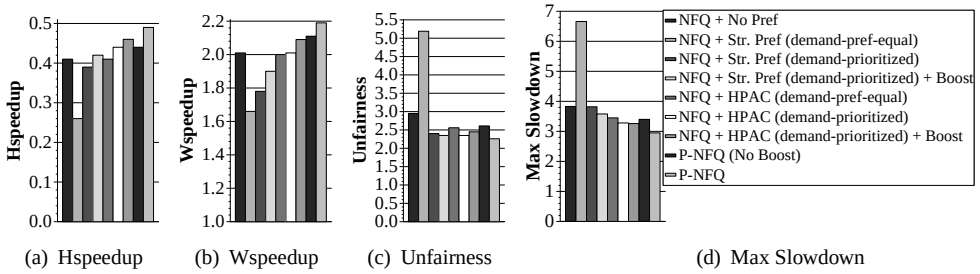
6.1. NFQ Results

Figures 5 (a)-(d) show average system performance and unfairness of a system using an NFQ memory scheduler in different configurations: with no prefetching, prefetching with and without prefetcher control, and with our proposed prefetch-aware NFQ. In the policies referred to as *demand-pref-equal*, demands and prefetches are treated equally in terms of prioritization based on earliest virtual finish time. In the *demand-prioritized* policy, demands are always prioritized over prefetches, and are scheduled earliest virtual finish time first. Figure 6 shows system performance for each of the 15 evaluated workloads for the nine configurations of NFQ that we evaluated. P-NFQ provides the highest

system performance and least unfairness among all the examined techniques. P-NFQ outperforms the best performing previous technique (NFQ + HPAC demand-prioritized) by 11%/8.6% (HS/WS) while reducing maximum slowdown by 9.9%. Several key observations are in order:

1. Figure 5 shows that in all cases (with or without prefetcher throttling), *demand-prioritized* has higher performance and lower maximum slowdown than *demand-pref-equal*. We conclude that as we explained in Section 4.2, if *all* prefetch requests are treated alike demand requests, wasted service given to useless prefetches leads to a worse-performing and less fair system than always prioritizing demands.

2. The last two bars in each of the subfigures of Figure 5 demonstrate **a key insight**: without intelligent prioritization of demand requests of memory non-intensive applications, system performance and fairness do not significantly improve *simply by* prioritizing accurate prefetches. Adding the demand boosting optimization to P-NFQ (with no boosting) improves performance by 10%/3.8% (HS/WS) and reduces maximum slowdown by 13.2% compared to just prioritizing accurate prefetches within NFQ's algorithm.



(a) Hspeedup (b) Wspeedup (c) Unfairness (d) Max Slowdown

Figure 5. Average system performance and unfairness on 4-core system with NFQ

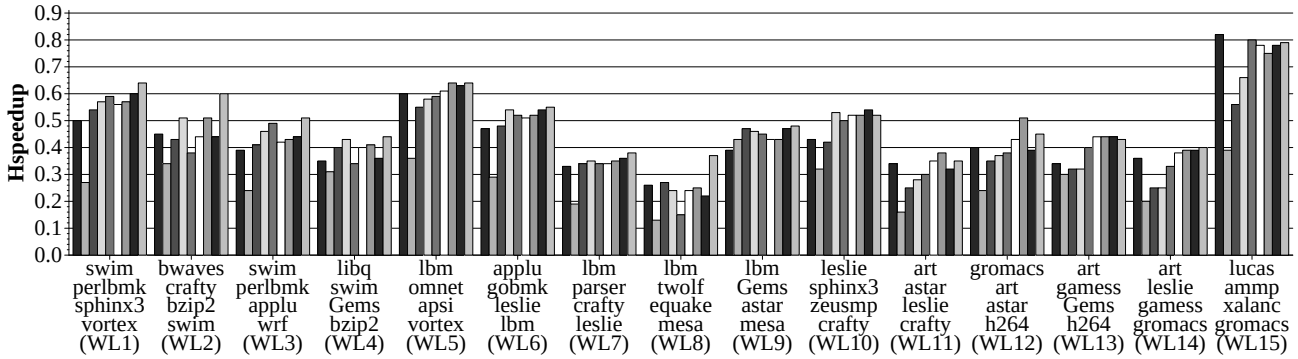


Figure 6. System performance (Hspeedup) for each of the 15 workloads (legend same as Figure 5)

3. Figures 5 (a)-(d) show that demand boosting improves system performance independent of the setup it is used with. Demand boosting alone improves the performance of demand-prioritized and prefetching with no throttling by 7.3%/6.7% (HS/WS). When used with demand-prioritized and HPAC, it improves performance by 3.3%/3.6% (HS/WS). However, demand boosting provides the best system performance and fairness when used *together* with our proposed P-NFQ which prioritizes requests based on virtual finish time first using prefetch accuracy feedback.

Note that demand boosting and considering prefetch accuracy information in prioritization decisions are synergistic techniques. Together they perform better than each one alone. We conclude that demand boosting is a general mechanism but is most effective when used together with resource management policies which take prefetcher accuracy into account in their prioritization rules.

6.2. PARBS Results

Figures 7 (a)-(d) show average system performance and unfairness of different prefetch-demand batching policies with and without prefetcher control. In *demand-pref-batching*, demands and prefetches are treated equally in PARBS's batch-forming (within the batches, demands are prioritized over prefetches because we find this to be better performing on average). In *demand-only-batching*, only demands are included in the batches. Figure 8 shows system performance for each of the 15 evaluated workloads for the nine configurations of PARBS that we evaluated. P-PARBS provides the highest system performance and the smallest unfairness among all of the techniques, improving system performance on average by 10.9%/4.4% (HS/WS) while reducing maximum slowdown by 18.4% compared to the combination of PARBS and HPAC with demand-only-batching.

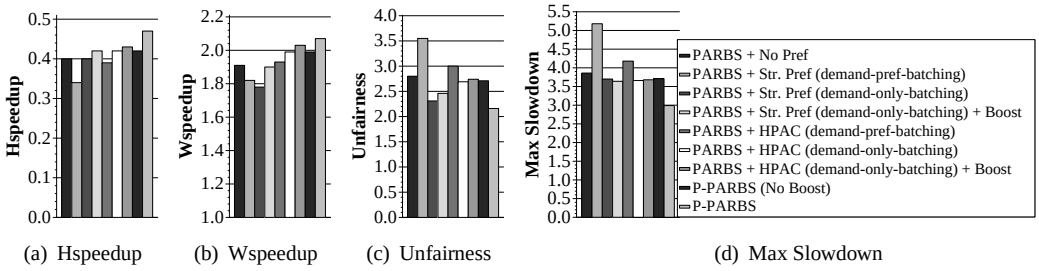


Figure 7. Average system performance and unfairness on 4-core system with PARBS

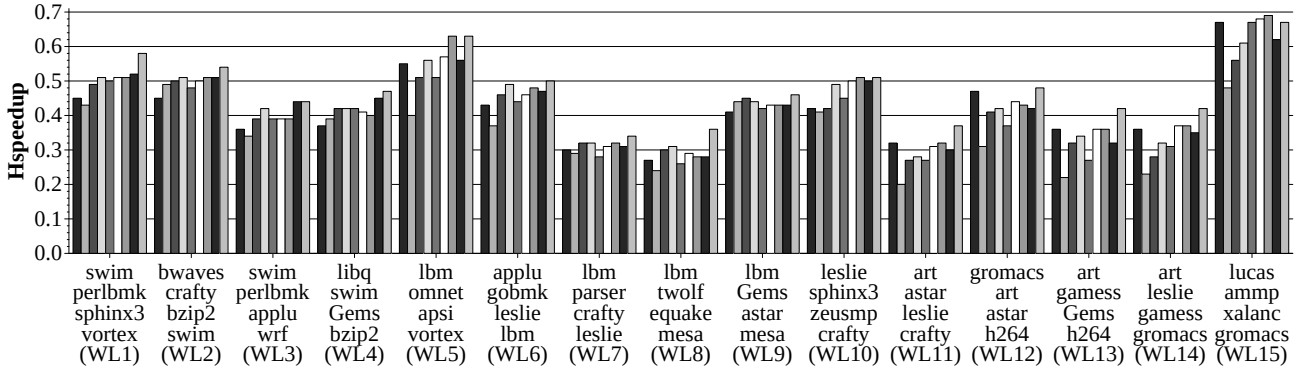


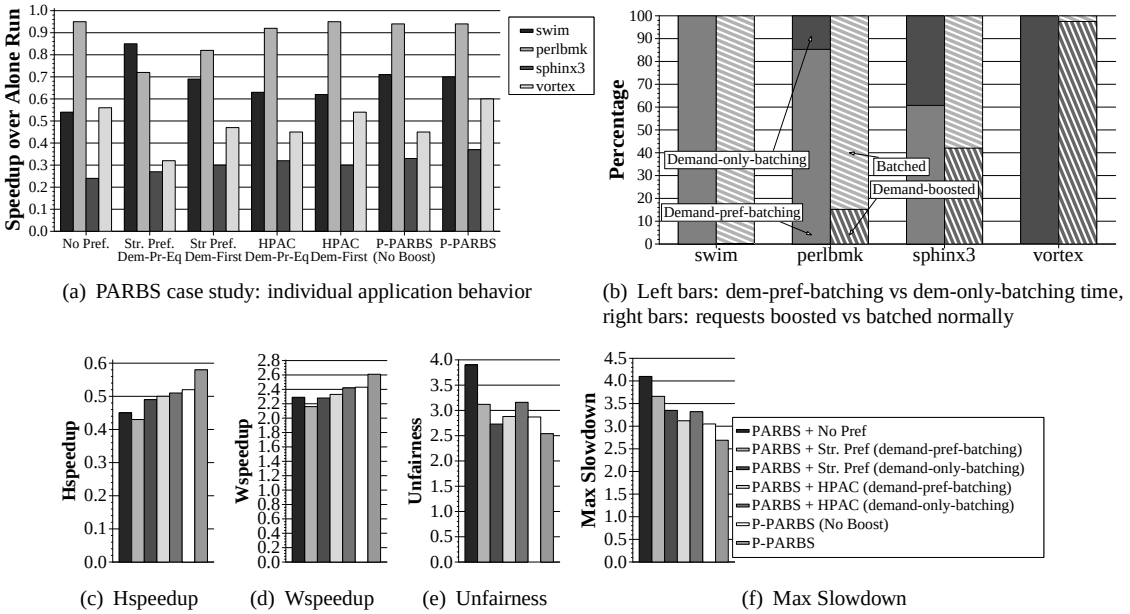
Figure 8. System performance (Hspeedup) for each of the 15 workloads (legend same as Figure 7)

6.2.1. Case Study The goal of this case study is to provide insight into how the mechanisms that we propose improve performance. It also shows in detail why *simply* prioritizing accurate prefetches in shared resource management techniques does not necessarily improve *system performance and fairness*. We examine a scenario where two memory

intensive and prefetch-friendly applications (*swim and sphinx3*) concurrently execute with two memory non-intensive applications (*perlbmk and vortex*). Figures 9 (a) and (c)-(f) show individual application performance and overall system behavior of this workload. Figure 9 (b) shows the dynamics of the mechanisms proposed for prefetch-aware PARBS. In Figure 9 (b), each application is represented with two bars. The left bar in each pair shows the percentage of time that *both* demands and prefetches from the corresponding application were included in P-PARBS's batches vs. the percentage of time that *only* demands were included. The right bar shows the percentage of all demand requests that were boosted into the batches by the demand-boosting mechanism vs. all other batched requests.

P-PARBS both performs significantly better and is much more fair than all the other evaluated techniques. This is due to the following two reasons:

1. Including useful prefetches of *swim* and *sphinx3* alongside demand requests in P-PARBS's batches allows these applications to make good use of their accurate prefetches and significantly improves their performance. Figure 9 (b) shows that *swim*'s and *sphinx3*'s prefetches are included in the batches for 100% and 60% of their execution times respectively. During these periods, *swim* and *sphinx3* also achieve better row buffer locality: their row buffer hits are increased by 90% and 27% respectively compared to the technique with the best system performance among the other techniques (HPAC demand-only-batching). In addition, *swim* and *sphinx3*'s prefetches become 8% and 11% more timely (not shown in the figure).



(a) PARBS case study: individual application behavior

(b) Left bars: dem-pref-batching vs dem-only-batching time, right bars: requests boosted vs batched normally

(c) Hspeedup (d) Wspeedup (e) Unfairness (f) Max Slowdown

Figure 9. PARBS case study

2. Boosting the demands of the prefetch insensitive and memory non-intensive application, *vortex*, allows it to get quick memory service and prevents it being delayed by the many requests batched for *swim* and *sphinx3*. Because

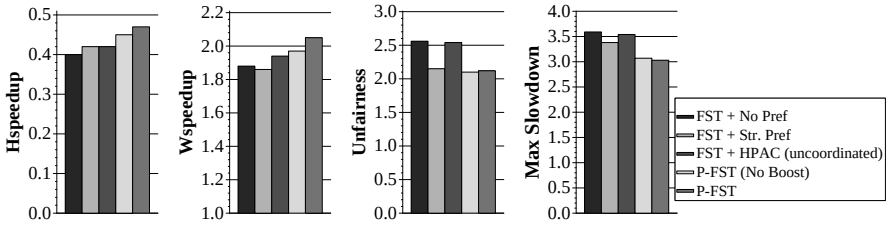
vortex's requests are serviced quickly, its performance increases. Also, since *vortex* is memory non-intensive, this boosting does not degrade other applications' performance significantly.

The last two sets of bars in Figure 9 (a) show the importance of the demand boosting optimization. When *swim*'s and *sphinx3*'s prefetches are included in the batches, *vortex*'s performance degrades if *demand boosting* is not used. This happens because of inter-core cache pollution caused by *swim* and *sphinx3*. Hence, even though *swim*'s and *sphinx3*'s performance improves significantly without boosting, overall system performance does not improve over the HPAC demand-only-batching (Figures 9 (c)-(d)). In contrast, with *demand boosting*, *vortex*'s performance also improves which enables P-PARBS to perform 13.3%/7.6% (HS/WS) better than the best previous approach while also reducing maximum slowdown by 17.8%.

6.3. FST Results

Figures 10 (a)-(d) show average system performance and unfairness of FST in the following configurations: without prefetching, with aggressive stream prefetching, with HPAC, and our proposed coordinated core and prefetcher throttling, i.e., P-FST (with and without demand boosting). Figure 11 shows system performance for each of the 15 evaluated workloads for the five configurations of FST that we evaluated. P-FST provides the highest performance and best fairness among the five techniques. Several observations are in order:

1. When prefetching with no throttling is used, in five of the workloads prefetcher-caused interference is noticeable and is left uncontrolled by FST. This results in large degradations in system performance of 5% or more (WL5, WL11, WL12, WL14, and WL15). In these workloads, FST does not detect the applications causing prefetcher interference to be *App-interfering*. Because of these workloads, prefetching with no throttling does not improve average system performance significantly compared to no prefetching as shown in Figure 10. This shows the need for explicit prefetcher throttling when prefetching is used with FST.



(a) Hspeedup (b) Wspeedup (c) Unfairness (d) Max Slowdown

Figure 10. Average system performance and unfairness on 4-core system with FST

2. When HPAC [2] and FST [3] are naively combined with no coordination, four of the 15 workloads lose significant prefetching performance (workloads WL1, WL3, WL4, and WL8). In such cases, HPAC throttles down some useful prefetchers unnecessarily. This happens due to: a) excessive throttling caused by HPAC's coarse classification of

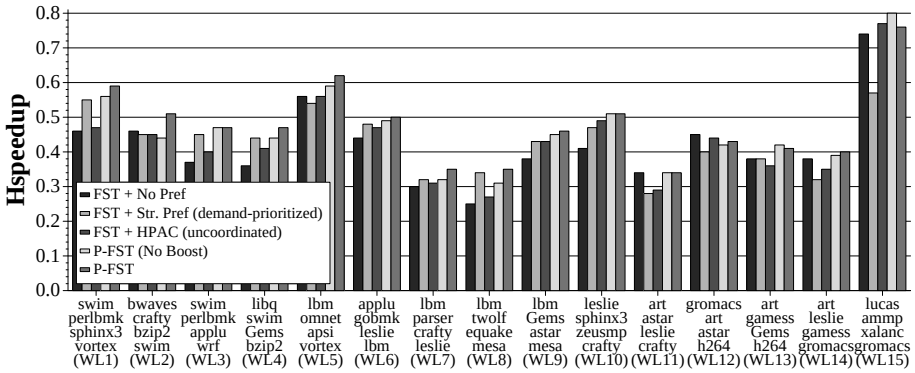


Figure 11. System performance (Hspeedup) for each of the 15 workloads

interference, and b) underestimation of prefetcher accuracy due to interference-unaware tracking of useful prefetches (described in Section 4.3.2). Unnecessary throttling makes the system more unfair compared to no prefetcher throttling. This happens when a prefetch-friendly application with the largest slowdown in the absence of prefetching is unnecessarily throttled. With no prefetcher throttling, such an application gains significant performance, which in turn reduces system unfairness. When HPAC throttles down the prefetchers of such applications too much, this fairness improvement is lost. We conclude that even though a naive combination of HPAC and FST improves average system throughput, this comes at the cost of increasing system unfairness significantly compared to no throttling.

3. Our P-FST technique (with demand-boosting) addresses the problems described above by coordinating prefetcher and core throttling, and improves performance by 11.3%/5.6% (HS/WS) while reducing maximum slowdown by 14.5% compared to the best performing of the other techniques (i.e. the uncoordinated FST and HPAC combination). Compared to the configuration with the least max slowdown, i.e. the combination of prefetching with no throttling and FST, P-FST with boosting performs 11.2%/10.3% (HS/WS) better while reducing maximum slowdown by 10.3%.

6.4. Effect on Homogeneous Workloads

Multi-core systems are sometimes used to run multiple copies of the same application in server environments. Table 3 shows system performance and fairness deltas of P-NFQ compared to NFQ + HPAC (demand-prioritized) for a prefetch friendly (four copies of sphinx3) and a prefetch unfriendly (four copies of astar) workload. Our proposal improves system performance and reduces max slowdown for the prefetch friendly workload, while it does not significantly affect the prefetch unfriendly one. In the prefetch friendly workload, prioritizing accurate prefetches improves each benchmark's performance by making timely use of those accurate prefetches. This is not possible if all prefetches are treated alike.

Four copies of sphinx3 (prefetch friendly)			Four copies of astar (prefetch unfriendly)		
Δ HS	Δ WS	Δ Max Slowdown	Δ HS	Δ WS	Δ Max Slowdown
7.9%	7.9%	-8.1%	-1%	-1%	0.5%

Table 3. Effect of our proposal on homogeneous workloads in system using NFQ memory scheduling

6.5. Sensitivity to System and Algorithm Parameters

Table 4 shows how P-NFQ performs compared to NFQ + HPAC (demand prioritized) on systems with two/four memory channels or 8MB/16MB shared last level caches. Even though using multiple memory channels reduces contention to DRAM, and using larger caches reduces cache contention, P-NFQ still performs significantly better while reducing maximum slowdown. We conclude that our mechanism provides performance benefits even on more costly systems with higher memory bandwidth or larger shared caches.

Single Channel			Dual Channel			Four Channel		
Δ HS	Δ WS	Δ Max Slowdown	Δ HS	Δ WS	Δ Max Slowdown	Δ HS	Δ WS	Δ Max Slowdown
11%	8.6%	-9.9%	5%	5.7%	-3.7%	4%	6.3%	0.7%
2MB Shared Cache			8MB Shared Cache			16MB Shared Cache		
Δ HS	Δ WS	Δ Max Slowdown	Δ HS	Δ WS	Δ Max Slowdown	Δ HS	Δ WS	Δ Max Slowdown
11%	8.6%	-9.9%	6.3%	5.3%	-9.1%	4.9%	3.9%	-6.6%

Table 4. Effect of our proposal on system using NFQ memory scheduling with different microarchitectural parameters

Figure 12 shows how sensitive the performance benefits of the techniques we propose (compared to the best previous technique in each case) are to the boosting threshold. For all shown thresholds, P-NFQ and P-FST show performance within 1% of that of the chosen threshold. For P-PARBS, this is the case for all values between 14 and 26. In P-PARBS, with thresholds less than 14, not enough requests from prefetch-unfriendly benchmarks get boosted. We conclude that the benefits of our mechanisms are not highly sensitive to the chosen threshold value.

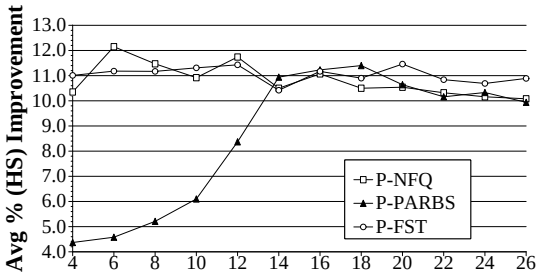


Figure 12. Sensitivity to boosting threshold

6.6. Hardware Cost

Table 5 shows the required storage of our mechanisms on top of each of the shared resource management techniques. Our mechanisms do not require any structures that are on the critical path of execution. Additionally, none of the structures we add/modify require large energy to access and none are accessed very often. As such, significant overhead is not introduced in terms of power.

7. Related Work

To our knowledge, this paper is the first to provide mechanisms that aim to improve both performance and fairness of fair shared resource management techniques in the presence of prefetching. The major contributions of this work are general mechanisms for how prefetches should be considered in the prioritization decisions of resource-based and source-based shared resource management. We have already provided extensive quantitative and qualitative com-

P-NFQ	Closed form for N cores (bits)	N=4(bits)
Boosting bits in memory request queue entries	32 x N	128
Counters for number of requests per core in memory request queue	8 x N	32
Total storage required for P-NFQ	40 x N	160
P-PARBS		
Counters for number of requests per core in memory request queue	8 x N	32
Total storage required for P-PARBS	8 x N	32
P-FST		
Boosting bits in memory request queue entries	32 x N	128
Counters for number of requests per core in memory request queue	8 x N	32
Prefetch bits in pollution filter used for coordinated core and prefetcher throttling	Pol. Filter Entries (2048) x N	8192
Total storage required for P-FST	2088 x N	8352

Table 5. Hardware cost of our proposed enhancements

parison to the three techniques that we apply our mechanisms to (NFQ [20], PARBS [19], and FST [3]) in previous sections. Here, we briefly discuss other related work in prefetch-aware DRAM controllers, shared resource management, and prefetch filtering.

7.1. Prefetch-Aware DRAM Controllers

Lee et. al. [12] propose using prefetch accuracy information to determine whether to prioritize demands over prefetches or to treat them equally in terms of memory scheduling. To our knowledge, this is the only prior work that deals with how prefetches should be dealt with in a shared resource. However, this work targets handling prefetches in a DRAM-throughput-oriented FR-FCFS scheduler that is not designed to provide fairness/QoS. Our paper makes two major contributions beyond this work. First, it is the first to address how prefetches should be considered in *fair/QoS*-capable memory scheduling techniques that are shown to provide significantly higher performance than throughput-oriented DRAM schedulers. Second, it provides generalized prefetch handling techniques not only for memory scheduling but also for a more general source throttling-based management technique that aims to manage multiple shared resources.

Lee et. al. [12] also observe the need for prioritizing demand requests of *applications which do not have accurate prefetchers*. They prioritize the demands of such applications over memory requests of prefetch friendly applications *which are not row buffer hits*. They place no condition on the memory intensity of the applications that are prioritized this way. Our demand boosting mechanism employs a more aggressive prioritization for a more limited class of requests. We prioritize demand requests of applications which are ***both*** not prefetch-friendly ***and*** memory non-intensive over ***all other*** memory requests. We find this to be more effective than their prioritization mechanism because it does not cause starvation or slowdown as it never boosts intensive applications' requests. In fact, in Figure 5, the P-NFQ (No boost) experiment includes the prioritization proposed by this prior work, on top of which, our P-NFQ improves performance by 10%. Note that the critical element that allows this aggressive boosting is that it is only performed for non-intensive phases of an application where it has a limited number of requests requiring service.

7.2. Other Shared Resource Management Techniques

Many previous papers deal with management of other shared resources such as caches [9, 24, 25, 7, 8, 21] or on-chip interconnect [14, 1, 5] to improve system performance and/or fairness. However, none deal with how prefetches should be intelligently dealt with in the mechanisms they propose. As such, this paper is orthogonal to prior work in shared resource management, and we believe the ideas we present in this paper can be used to enhance other shared resource management techniques in the presence of prefetching.

7.3. Prefetch Filtering

Prior papers on prefetch filtering propose techniques to detect and eliminate prefetches that are found to be useless in the past [29, 31, 15]. However, conservative prefetch filtering techniques cannot filter out all useless prefetches, yet aggressive ones can remove many useful prefetches [12]. We find that our proposals are complementary to prefetch filtering. For example, P-NFQ increases system performance by 11.1%/8.2% (HS/WS) while reducing max slowdown by 10.9% on a baseline where a state-of-the-art hardware filtering mechanism [31] is used with NFQ + HPAC.

8. Conclusion

This paper demonstrates a new problem in CMP designs: state-of-the-art fair shared resource management techniques, which significantly enhance performance/fairness in the absence of prefetching, can largely degrade performance/fairness in the presence of prefetching. To solve this problem, we introduce general mechanisms to effectively handle prefetches in multiple types of resource management techniques.

We develop three major new ideas to enable prefetch-aware shared resource management. We introduce the idea of *demand boosting*, a mechanism that eliminates starvation of applications that are not prefetch-friendly yet memory non-intensive, thereby boosting performance and fairness of any type of shared resource management. We describe how to intelligently prioritize demands and prefetches within the underlying fair management techniques. We develop new mechanisms to coordinate the actions of prefetcher and core throttling mechanisms to make synergistic decisions. To our knowledge, this is the first paper that deals with prefetches in shared multi-core resource management, and enables such techniques to be effective and synergistic with prefetching.

We apply these new ideas to three state-of-the-art multi-core shared resource management techniques. Our extensive evaluations show that our proposal significantly improves system performance and fairness of two fair memory scheduling techniques and one source-throttling-based shared memory system management technique (by more than 10% in 4-core systems), and makes these techniques effective with prefetching. We conclude that our proposal can be a low-cost and effective solution that enables the employment of both prefetching and shared resource management together in future multi-core systems, thereby ensuring future systems can reap the performance and fairness benefits

of both ideas together.

References

[1] R. Das et al. Application-aware prioritization mechanisms for on-chip networks. In *MICRO*, 2009.

[2] E. Ebrahimi et al. Coordinated control of multiple prefetchers in multi-core systems. In *MICRO*, 2009.

[3] E. Ebrahimi et al. Fairness via source throttling: A configrable and high-performance fairness substrate for multi-core memory systems. In *ASPLOS*, 2010.

[4] R. Gabor, S. Weiss, and A. Mendelson. Fairness and throughput in switch on event multithreading. In *MICRO-39*, 2006.

[5] B. Grot et al. Preemptive virtual clock: A flexible, efficient, and cost-effective QoS scheme for networks-on-a-chip. In *MICRO*, 2009.

[6] G. Hinton et al. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Feb. 2001. Q1 2001 Issue.

[7] L. R. Hsu et al. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *PACT-15*, 2006.

[8] R. Iyer et al. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS'07*, June 2007.

[9] S. Kim et al. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.

[10] Y. Kim et al. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA-16*, 2010.

[11] H. Q. Le et al. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51:639–662, 2007.

[12] C. J. Lee et al. Prefetch-aware DRAM controllers. In *MICRO-41*, 2008.

[13] C. J. Lee et al. Improving memory bank-level parallelism in the presence of prefetching. In *MICRO-42*, 2009.

[14] J. W. Lee et al. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *ISCA-35*, 2008.

[15] W.-F. Lin, S. K. Reinhardt, D. Burger, and T. R. Puzak. Filtering superfluous prefetches using density vectors. In *ICCD*, 2001.

[16] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*, 2001.

[17] Micron. *Datasheet: 2Gb DDR3 SDRAM, MT41J512M4 - 64 Meg x 4 x 8 banks, http://download.micron.com/pdf/datasheets/dram/ddr3*.

[18] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO-40*, 2007.

[19] O. Mutlu and T. Moscibroda. Paralleism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA-35*, 2008.

[20] K. J. Nesbit et al. Fair queuing memory systems. In *MICRO-39*, 2006.

[21] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA-34*, June 2007.

[22] J. Owen and M. Steinman. Northbridge architecture of AMD's Griffin microprocessor family. *IEEE Micro*, 28(2), 2008.

[23] H. Patil et al. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.

[24] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. 2006.

[25] N. Rafique et al. Architectural support for operating system-driven CMP cache management. In *PACT-15*, 2006.

[26] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000.

[27] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *ASPLOS-IX*, 2000.

[28] S. Srinath et al. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.

[29] V. Srinivasan et al. A static filter for reducing prefetch traffic. Technical Report CSE-TR-400-99, University of Michigan, 1999.

[30] J. Tendler et al. POWER4 system microarchitecture. *IBM Technical White Paper*, Oct. 2001.

[31] X. Zhuang and H.-H. S. Lee. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *ICPP-32*, 2003.