Coordinated Control of Multiple Prefetchers in Multi-Core Systems

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

†Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, cjlee, patt}@ece.utexas.edu

§Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu

ABSTRACT

Aggressive prefetching is very beneficial for memory latency tolerance of many applications. However, it faces significant challenges in multi-core systems. Prefetchers of different cores on a chip multiprocessor (CMP) can cause significant interference with prefetch and demand accesses of other cores. Because existing prefetcher throttling techniques do not address this prefetcher-caused inter-core interference, aggressive prefetching in multi-core systems can lead to significant performance degradation and wasted bandwidth consumption.

To make prefetching effective in CMPs, this paper proposes a low-cost mechanism to control prefetcher-caused inter-core interference by dynamically adjusting the aggressiveness of multiple cores' prefetchers in a coordinated fashion. Our solution consists of a hierarchy of prefetcher aggressiveness control structures that combine per-core (local) and prefetcher-caused inter-core (global) interference feedback to maximize the benefits of prefetching on each core while optimizing overall system performance. These structures improve system performance by 23% while reducing bus traffic by 17% compared to employing aggressive prefetching and improve system performance by 14% compared to a state-of-the-art prefetcher aggressiveness control technique on an eight-core system.

Categories and Subject Descriptors: C.1.0 [Processor Architectures]: General; C.5.3 [Microcomputers]: Microprocessors; C.1.2 [Multiple Data Stream Architectures (Multiprocessors)]

General Terms: Design, Performance.

Keywords: Prefetching, multi-core, memory systems, feedback control.

1. INTRODUCTION

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. By doing so, prefetching can hide the latency of a memory access since the processor either does not incur a cache miss for that access or incurs a cache miss that is satisfied earlier because the prefetch request already started the memory access.

In a chip-multiprocessor (CMP) system, cores share memory system resources beyond some level in the memory hierarchy. Bandwidth to main memory and a shared last level cache are two important shared resources in almost all CMP designs. Aggressive prefetching on different cores of a CMP, although very beneficial for memory latency tolerance on many applications when they are run alone, can ultimately lead to 1) significant system performance degradation and bandwidth waste compared to no prefetching, or 2) relatively small system performance improvements with prefetching. This is a result of the following types of prefetcher-caused inter-core interference in shared resources: 1) *prefetch-prefetch interference*: prefetches from one core can delay or displace prefetches from another core by causing contention for memory bandwidth and cache space, and 2) *prefetch-demand interference:* prefetches from one core can either delay demand (load/store) requests from another core or displace the other core's demand-fetched blocks from the shared caches. Our goal in this paper is to develop a hardware framework that enables large performance improvements from prefetching in CMPs by significantly reducing prefetcher-caused inter-core interference.

Prefetcher-caused inter-core interference can be somewhat reduced if the prefetcher(s) on each core are individually made more accurate. Previous work [31, 8, 27, 13, 6] proposed techniques to throttle the aggressiveness or increase the accuracy of prefetchers. As a side effect, such techniques can also reduce prefetcher-caused inter-core interference compared to a system that enables aggressive prefetching without any prefetcher control. However, proposed prefetcher throttling techniques [8, 27, 6] only use feedback information *local* to the core the prefetcher resides on. Mechanisms that attempt to reduce the negative effects of aggressive prefetching by filtering useless prefetch requests [13, 31] also operate independently on each core's prefetch requests. Not taking into account feedback information about the amount of prefetcher-caused *inter*-core interference is a major shortcoming of previous techniques. We call this feedback information *global (or system-wide) feedback*.

Why is *global feedback* important? Figure 1 compares the performance improvement obtained by independently throttling the prefetcher on each core using state-of-the-art feedback-directed prefetching (FDP) [27] to that obtained by an unrealizable system that, in addition to using FDP, *ideally* eliminates all prefetcher-caused inter-core interference in shared memory resources. To model the ideal system, for each core we eliminated all memory request buffer entry conflicts, memory bank conflicts, row buffer conflicts, and cache pollution caused by another core's prefetcher, but we model all similar interference effects caused by the same core's prefetcher. This experiment was performed for 32 multiprogrammed workloads on a 4-core system.[1] Independently throttling each prefetcher using FDP improves performance by only 4%. In contrast, if all prefetcher-caused inter-core interference were ideally eliminated, performance would improve by 56% on average. Hence, significant performance potential exists for techniques that con-

MICRO'09, December 12–16, 2009, New York, NY, USA.
Copyright 2009 ACM 978-1-60558-798-1/09/12 ...$10.00.

[1] The reported average is over all 32 workloads, but only 14 of them are shown here. These are the same workloads shown in Figure 7, which constitute four classes of workloads analyzed in Section 5.2. Potential performance improvements for the other 18 workloads were evaluated and showed similar results.

trol prefetcher-caused inter-core interference. Moreover, we find that, in some workloads, independently throttling the prefetcher on each core degrades system performance because it blindly increases the aggressiveness of accurate prefetchers. However, using global feedback, coordinated and collective decisions can be made for prefetchers of different cores, leading to significant performance and bandwidth-efficiency improvements, as we show in this paper.

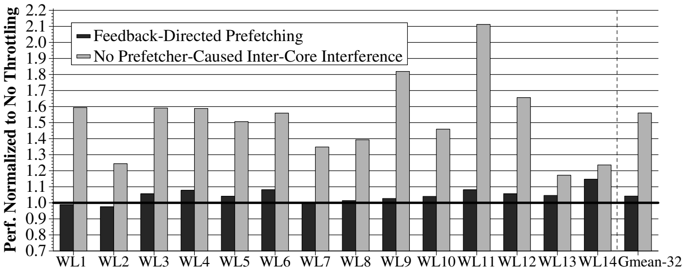


Figure 1: System performance improvement of ideally eliminating prefetcher-caused inter-core interference vs. feedback-directed prefetching (1.0 is the baseline performance with no throttling; performance measured in harmonic speedup, see Section 4)

Basic Idea We develop a mechanism that controls the aggressiveness of the system's prefetchers in a hierarchical fashion, called Hierarchical Prefetcher Aggressiveness Control (HPAC). HPAC dynamically adjusts the aggressiveness of each prefetcher in two ways: *local* and *global*. The local decision attempts to maximize the local core's performance by taking into account only local feedback information, similar to previous prefetcher throttling mechanisms [8, 27, 6]. The global mechanism can override the local decision by taking into account effects and interactions of different cores' prefetchers when adjusting each one's aggressiveness. The key idea is that if prefetcher-caused interference in the shared cache and memory bandwidth is estimated to be significant, the global control system enforces a throttling decision that is best for overall system performance rather than allowing the local control to make a less-informed decision that may degrade overall system performance.

Summary of Evaluation We evaluate our technique on both 4-core and 8-core CMP systems. We find that as the number of cores in a CMP increases, the benefits of our technique also increase. Experimental results across a wide variety of workloads on an 8-core CMP show that our proposed *Hierarchical Prefetcher Aggressiveness Control* technique improves average system performance by 23%/14% and reduces memory bus traffic by 17%/3.2% compared respectively to a system that enables aggressive stream prefetching on all cores without any throttling and a system with a state-of-the-art aggressiveness control mechanism [27] enabled individually for each prefetcher.

Contributions To our knowledge, our proposal is the first comprehensive solution to dynamic control of prefetcher aggressiveness that uses system-wide inter-core prefetcher interference information to maximize overall system performance. We make the following contributions:

1. We show that uncoordinated, local-only prefetcher control can lead to significant system performance degradation compared to no prefetching even though it might make a seemingly "correct" *local* decision for each core's prefetcher in an attempt to maximize that particular core's performance.

2. We propose a low-cost mechanism to improve the performance and bandwidth-efficiency of prefetching and make it effective in CMPs. The proposed mechanism uses a hierarchical approach to prefetcher aggressiveness control. It optimizes overall system performance with *global control* using inter-core prefetcher interference feedback from the shared memory system, while maximizing prefetcher benefits on each core with *local control* using per-core feedback.

3. We demonstrate that our approach i) is orthogonal to various memory scheduling policies (Section 5.3), ii) significantly improves and is orthogonal to various state-of-the-art local-only prefetcher throttling (Section 5.5) and pollution filtering (Section 6.2) techniques, iii) is applicable to multiple types of prefetchers per core (Section 5.5), and iv) is orthogonal to previously proposed fairness and quality of service techniques (Section 5.4) in shared multi-core resources.

2. BACKGROUND AND MOTIVATION

We briefly describe relevant previous research on prefetcher aggressiveness control, since our proposal builds on this prior work. We also describe the shortcomings of these prefetcher control mechanisms and provide insight into the potential benefits of reducing prefetcher-caused inter-core interference using coordinated control of multiple prefetchers.

2.1 Previous Techniques for Controlling Prefetcher Aggressiveness

Almost all prefetching algorithms contain a design parameter determining their aggressiveness [12, 1, 11, 4, 22]. For example, in many stream or stride prefetcher designs, *prefetch distance* and *prefetch degree* are two parameters that define how aggressive the prefetcher is [27]. Prefetch distance refers to how far ahead of the demand miss stream the prefetcher can send requests, and prefetch degree determines how many requests the prefetcher issues at once.

In applications where a prefetcher's requests are accurate and timely, a more aggressive prefetcher can achieve higher performance. On the other hand, in applications where prefetching is not useful, aggressive prefetching can lead to large performance degradation due to cache pollution and wasted memory bandwidth, and higher power consumption due to increased off-chip accesses. To reduce these problems, there have been proposals for controlling a prefetcher's aggressiveness based on its usefulness. For example, Feedback-Directed Prefetching (FDP) [27] is a prefetcher throttling technique that collects feedback local to a single prefetcher (i.e., the prefetcher's accuracy, timeliness, and pollution on the local core's cache) and adjusts its aggressiveness to reduce the negative effects of prefetching. Coordinated throttling [6] is a mechanism to adjust the aggressiveness levels of more than one type of prefetcher in a hybrid prefetching mechanism. It combines feedback from multiple prefetchers of a single core and makes a decision on how aggressive each prefetcher should be. Its goal is to maximize the contribution of each prefetcher to the single core's performance while reducing negative effects of useless prefetches. Neither of these prior works take into account information from other cores in a multi-core system.

2.2 Shortcomings of Local-Only Prefetcher Control

Prior approaches to controlling prefetcher aggressiveness that use only information local to each core can make incorrect decisions from a system-wide perspective. Consider the example in Figures 2 and 3. In the 4-core workload shown, employing aggressive stream prefetching increases the performance of *swim* and *lbm* (by 86% and 30%) and significantly degrades the performance of *crafty* and *bzip2* (by 57% and 35%). This results in an overall reduction in system performance of 5% (harmonic speedup - defined in Section 4) and an increase in bus traffic of 10% compared to no prefetching. As Figure 2 shows, with FDP, applications independently gain some performance, however, even with these gains, system performance still degrades by 4% and bus traffic increases by 7% compared to no prefetching. In contrast, our HPAC proposal makes a coordinated decision for the aggressiveness of multiple prefetchers.

As a result, system performance increases by 19.1% (harmonic speedup) while bus traffic increases by only 3.5% compared to no prefetching as shown in Figure 3.

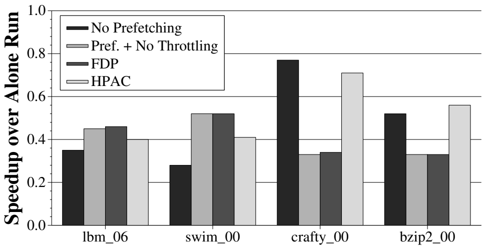
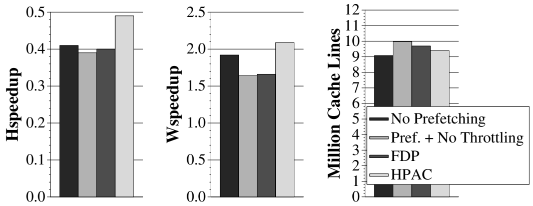


Figure 2: Speedup of each application w.r.t. when run alone



(a) Hspeedup (b) Wspeedup (c) Bus Traffic

Figure 3: System performance

The key to this performance improvement is throttling down of *swim*'s and *lbm*'s prefetchers. When these prefetchers are very aggressive, they cause significant pollution for other applications in the shared cache and cause high contention for DRAM banks. HPAC detects the interference caused by *swim*'s and *lbm*'s aggressive prefetchers. As a result, even though FDP *incorrectly* decides to throttle up the prefetchers (because the prefetchers are very accurate), HPAC throttles down the prefetchers using global feedback on interference. Doing so results in a loss of *swim*'s and *lbm*'s performance compared to aggressive prefetching. However, this allows *bzip2* to gain performance with prefetching, which was not realizable for this application with no throttling or with FDP, and significantly reduces *crafty*'s performance degradation. Overall, HPAC enables significant performance improvement due to prefetching which cannot be obtained with no throttling or FDP.

The key insight is that a control system that is aware of prefetcher-caused inter-core interference in the shared memory resources can keep an *accurate* but overly aggressive prefetcher in check, whereas a local-only control scheme would allow it to continue to interfere with other cores' memory requests and cause overall system performance degradation.

Our goal: In this paper, we aim to provide a solution to prefetcher control to significantly improve the performance of prefetching and make it effective on a large variety of workloads in CMP systems. Our HPAC mechanism does exactly that by combining system-wide and per-core feedback information to throttle the aggressiveness of multiple prefetchers of different cores in a coordinated fashion.

3. HIERARCHICAL PREFETCHER AGGRESSIVENESS CONTROL (HPAC)

The Hierarchical Prefetcher Aggressiveness Control (HPAC) mechanism consists of *local* and *global* control structures. The two structures have fundamentally different goals and are hence designed very differently as explained in detail below.

3.1 Local Aggressiveness Control Structure

The local control structure adjusts the aggressiveness of the prefetcher(s) of each core with the sole goal of maximizing the performance of that core. This structure is not aware of the overall system picture and the interference between memory requests of different cores. Prior research [27, 6] proposed such structures. Such previously proposed structures or other novel structures that determine the aggressiveness of a single core's prefetcher(s) are orthogonal to the ideas presented in this paper and could be incorporated as the local control mechanism of the system proposed here. In fact, we evaluate the use of two previous proposals, FDP [27] and coordinated throttling [6], as our local control structure in Section 5.5.

3.2 Global Aggressiveness Control Structure

The *global* aggressiveness control structure keeps track of prefetcher-caused inter-core interference in the shared memory system. The global control can accept or override decisions made by each local control structure with the goal of increasing overall system performance and bandwidth efficiency.

3.2.1 Terminology

We first provide the terminology we will use to describe the global aggressiveness control. For our analysis we define the following terms, which are used as global feedback metrics in our mechanism:

Accuracy of a Prefetcher for Core i **-** ACC_i**:** The fraction of prefetches sent by core i's prefetcher(s) that were used by subsequent demand requests.

Pollution Caused by Core i**'s prefetcher(s) -** POL_i**:** The number of demand cache lines of all cores j evicted by core i's ($j \neq i$) prefetches that are requested subsequent to eviction.[2] This indicates the amount of disturbance a core's prefetches cause in the cache to the demand-fetched blocks of other cores.

Bandwidth Consumed by Core i **-** BWC_i**:** The sum of the number of DRAM banks servicing requests (demand or prefetch) from core i every cycle.

Bandwidth Needed by Core i **-** BWN_i**:** The sum of the number of DRAM banks that are busy every cycle servicing requests (demand or prefetch) from cores j when there is a request (demand or prefetch) from core i ($j \neq i$) queued for that bank in that cycle. This indicates the number of outstanding requests of a core that would have been serviced in the DRAM banks had there been no interference from other cores.

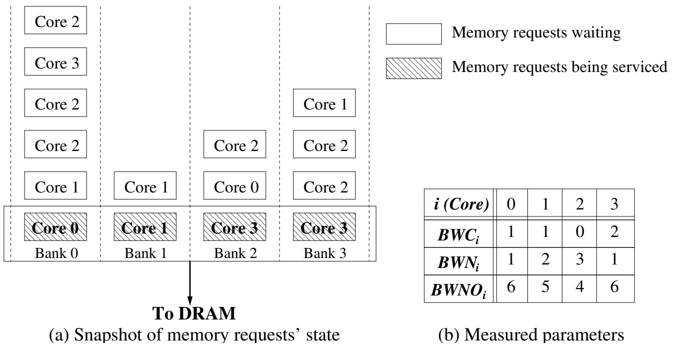
Bandwidth Needed by Cores Other than Core i **-** $BWNO_i$**:** The sum of the needed bandwidth of all cores except core i for which the prefetcher throttling decision is being made. Therefore,

$$BWNO_i = \sum_{j=0,\ j\neq i}^{N-1} BWN_j, \qquad N : Number\ of\ cores$$

Note that the global feedback metrics we define include information on interference affecting *both* demand and prefetch requests of different cores.

Example: Figure 4 illustrates the concepts of bandwidth consumption and bandwidth need. Figure 4(a) does not show many details of the DRAM subsystem but provides a framework to better understand the definitions above. It shows a snapshot of the DRAM subsystem with four requests being serviced by the different DRAM banks while other requests are queued waiting for those banks to be released. Based on the definitions above, the "Bandwidth consumed by a core" (BWC_i) and "Bandwidth needed by a core" (BWN_i) counts of the four different cores are incremented with the values shown in Figure 4(b) in the cycle the snapshot was taken. We focus on the increments for BWN of cores 1 and 2 to point out some subtleties. Core 1 has one request waiting for bank 0, one waiting for bank 1, and one waiting for bank 3. However, when calculating BWN

[2] Please note this definition is different from that used by Srinath et al. [27] for pollution caused by inaccurate prefetches on the same core's demands.



(a) Snapshot of memory requests' state

i (Core)	0	1	2	3
BWC_i	1	1	0	2
BWN_i	1	2	3	1
$BWNO_i$	6	5	4	6

(b) Measured parameters

Figure 4: Example of how to measure BWC_i, BWN_i, and $BWNO_i$

of core 1, only the requests waiting for bank 0 and bank 3 are accounted for. If there was no interference in the system, and if core 1 was the only core using the shared resources, the request from core 1 in the queue for bank 1 would still have had to wait. Hence, the BWN count for core 1 is incremented by 2 in this cycle. Core 2 has three requests waiting for bank 0, one request waiting for bank 2 and two requests waiting for bank 3. However, if there was no interference present only one of the three requests waiting for bank 0, the request waiting for bank 2, and one of the requests waiting for bank 3 could have been serviced in the cycle shown by the snapshot. Hence, the BWN count for core 2 is incremented by 3 in this cycle.

Intuitively, BWC corresponds to the amount of shared bandwidth used by a particular core. A core with high BWC can potentially delay other cores' simultaneous access to the shared DRAM banks and have a negative impact on their memory access latencies. BWN corresponds to the amount of bandwidth a core is *denied* due to interference caused by other cores in the system. A core might be causing interference for other cores if the sum of BWN of other cores grows too large (i.e., $BWNO$ of the core is too large).

3.2.2 Global Control Mechanism

In this section we explain how the feedback defined above is used to implement the global control mechanism. We refer to the prefetcher being throttled as the *target* prefetcher. When making a decision to allow or override the decision of a prefetcher's local control, global control needs to know: i) how accurate that prefetcher is, and ii) how much interference the prefetcher is causing for other cores in the system. In our proposed solution, we use the following parameters to identify how much interference the prefetcher of core i is generating for other cores in the shared resources: 1) the bandwidth consumed by core i (BWC_i), 2) the pollution caused by the prefetcher(s) of core i on other cores' demand requests (POL_i), and 3) the bandwidth needed by the other cores' requests (both prefetch and demand) ($BWNO_i$). Parameter 1, BWC_i, indicates the potential for increased interference with other cores due to the bandwidth consumption of core i. A high BWC_i indicates that core i will potentially cause interference if the target prefetcher's aggressiveness is not kept in check. Parameters 2 and 3 indicate the existence of such interference in the form of high bandwidth needs of other cores ($BWNO_i$) or cache pollution experienced by other cores (POL_i). When $BWNO_i$ or POL_i has a high value, high interference has been detected, and hence measures are required to reduce it.

Our global control mechanism is an interval-based mechanism that gathers the described feedback parameters during each interval. At the end of an interval, global control uses the collected feedback to allow or override the decision made by the target prefetcher's local control using the following principles.

Principle 1. When the target prefetcher shows low pollution (low POL_i):

(a) If the accuracy of the prefetcher is low[3] and other cores need a lot of bandwidth (i.e., $BWNO$ of the core is high), then override the local control's decision and throttle down. **Rationale:** this state indicates that an inaccurate prefetcher's requests have caused bandwidth interference that is negatively affecting other cores. Hence, the inaccurate prefetcher should be throttled down to reduce the negative impact of its inaccurate prefetches on other cores.

(b) If the accuracy of the prefetcher is low and the prefetcher's core is consuming a large amount of bandwidth (i.e., BWC of the core is high), our global control mechanism allows the local decision to affect the prefetcher only if the local control decides to throttle down. Otherwise, global control leaves the aggressiveness at its current level. **Rationale:** this is a state where interference can potentially worsen because the high bandwidth consumption of an inaccurate prefetcher's core can result in high bandwidth needs for other cores.

(c) If the prefetcher is highly accurate, then allow the local control to decide the aggressiveness of the prefetcher. **Rationale:** if a highly accurate prefetcher is not polluting other cores' demand requests (i.e., POL of the core is low), it should be given the opportunity to increase the performance of its local core.

Principle 2. When the target prefetcher is polluting other cores (high POL_i):

(a) If the accuracy of the prefetcher is low, then override the local control's decision and throttle down. **Rationale:** if an inaccurate prefetcher's requests pollute the demands of other cores, it could be negatively affecting system performance.

(b) If the target prefetcher is highly accurate, then allow the local decision to proceed if there are no other signs of interference (both BWC and $BWNO$ of the core are low). **Rationale:** if the bandwidth needs of all cores are observed to be low, the high pollution caused by the target prefetcher is likely not affecting the performance of other cores.

(c) If either bandwidth consumed (BWC) by the target prefetcher's core is high or other cores need a lot of bandwidth ($BWNO$ is high), then only allow the local decision to affect the prefetcher if it decides to throttle down, otherwise leave aggressiveness at its current level. **Rationale:** even though the prefetcher is accurate, it is showing more than one sign of interference which could be damaging overall system performance.

Rules used for global aggressiveness control: Table 1 shows the rules used by the global control structure. There is one case in this table that does not follow the general principles described above, case 14. In this case, interference is quite severe even though the target prefetcher is highly accurate. The target prefetcher's core is consuming a lot of bandwidth and is polluting other cores' demands while other cores have high bandwidth needs. Due to high interference detected by multiple feedback parameters, reducing prefetcher aggressiveness is desirable. The decision based on general principles would be: "Allow local decision only if it throttles down," which is not strong enough to alleviate this very high interference scenario. Therefore, we treat case 14 as an exception to the aforementioned principles and enforce a throttle-down with global control.

Classification of global control rules: We group the cases of Table 1 into three main categories classified based on the intensity of the interference detected by each case.

1) *Severe interference scenario:* Cases 3, 8, 9, 10 and 14 fall into this category. In these cases, the goal of the global control is to reduce the detected severe interference by reducing the number of prefetch requests generated by the interfering prefetchers. When the target prefetcher is inaccurate, and there is high bandwidth need from other cores (case 3), when an inaccurate

[3] Note that the local and global control structures can have separate thresholds to categorize an accuracy value as *low* or *high*.

Case	Info from core i			Info from other cores	Decision	Rationale
	Acc_i	BWC_i	POL_i	$BWNO_i$		
1	Low	Low	Low	Low	Allow local decision	No interference
2	Low	High	Low	Low	Allow local throttle down	1(b)
3	Low	-	Low	High	Global enforces throttle down	1(a)
4	High	Low	Low	Low	Allow local decision	1(c)
5	High	High	Low	Low	Allow local decision	1(c)
6	High	Low	Low	High	Allow local decision	1(c)
7	High	High	Low	High	Allow local decision	1(c)
8	Low	Low	High	Low	Global enforces throttle down	2(a)
9	Low	High	High	Low	Global enforces throttle down	2(a)
10	Low	-	High	High	Global enforces throttle down	2(a)
11	High	Low	High	Low	Allow local decision	2(b)
12	High	High	High	Low	Allow local throttle down	2(c)
13	High	Low	High	High	Allow local throttle down	2(c)
14	High	High	High	High	Global enforces throttle down	Very high interference

Table 1: Global control rules - ACC_i: Accuracy of prefetcher, BWC_i: Consumed bandwidth, POL_i: Pollution imposed on other cores, and $BWNO_i$: Sum of needed bandwidth of other cores

prefetcher is polluting (cases 8, 9 and 10), or when a prefetcher consumes high bandwidth, is polluting, and causes high bandwidth needs on other cores (case 14), prefetcher aggressiveness should be reduced regardless of the local decision. After the prefetcher has been throttled down and the detected interference has become less severe (by either improved accuracy of the target prefetcher, reduced pollution for other cores, or reduced bandwidth needs of other cores), the global throttling decisions for this prefetcher will be relaxed. This will allow the prefetcher to either not be throttled down further or throttled up based on local control's future evaluation of the prefetcher's behavior.

2) *Borderline interference scenario:* Cases 2, 12 and 13 fall into this category. In these cases, the global control's goal is to prevent the prefetcher from transitioning into a severe interference scenario, by either not throttling the prefetcher or throttling it down at the request of the local control. When an inaccurate prefetcher consumes high bandwidth but is not polluting (case 2), or when an accurate polluting prefetcher either consumes high bandwidth or causes high bandwidth need for other cores (cases 12 and 13), the prefetcher should not be throttled up as a result of the local control structure's decision.

3) *No interference scenario or moderate interference by an accurate prefetcher:* All other cases fall in this category. In these cases, either there is no interference or an accurate prefetcher has moderate interference. As explained in the general principles, in these cases, the prefetchers' aggressiveness is decided by the local control structures optimizing for highest performance in each core. We empirically found that high prefetcher accuracy can overcome the negative effects of moderate interference (cases 5, 6, 7 and 11) and therefore the local decision is used.

In Section 5.2.2, we present a detailed case study to provide insight into how prefetcher-caused inter-core interference hampers system performance and how HPAC improves performance significantly by reducing such interference.

3.2.3 Handling Multiple Prefetchers on Each Core

HPAC can seamlessly support systems with multiple types of prefetchers per core. In such systems, where speculative requests from different prefetchers can potentially increase prefetcher-caused inter-core interference, having a mechanism that takes such interference into account is even more important. In a system with multiple prefetchers on each core, the system-level feedback information referred to in Table 1 for each core corresponds to *all* the prefetchers on that core as a whole. For example, accuracy is the overall accuracy of all prefetchers on that core. Similarly, pollution is the overall shared cache pollution caused by all prefetchers from that core.

Note that prior research on intra-core prefetcher management [6] is orthogonal to the focus of this paper. In HPAC, when the local aggressiveness control corresponding to each core makes a decision for one of the prefetchers on that core, the global control allows or overrides that decision based on the effects and interactions of other cores' prefetchers.

3.2.4 Support for System-Level Application Priorities

So far, we have assumed concurrently running applications are of equal priority and hence are treated equally. However, system software (operating system or virtual machine monitor) may make policy decisions prioritizing certain applications over others in a multi-programmed workload. We seamlessly extend HPAC to support such priorities: 1) separate threshold values can be used for each concurrently-running application, 2) these separate threshold values are configurable by the system software using privileged instructions. To prioritize a more important application within HPAC, the system software can simply set a higher threshold value for $BWNO_i$, POL_i, and BWC_i and a lower threshold value for Acc_i for that application. By doing so, HPAC allows a more important application's prefetcher to cause more interference for other applications if doing so improves the more important application's performance.

3.2.5 Optimizing Threshold Values and Decision Set

Genetic algorithms [9] can be used to optimize the threshold value set or decision set of HPAC at design time. We implemented and evaluated a genetic algorithm for this purpose. We found that the improvements obtained by optimizing the decision set were not significant, but a 5% average performance improvement on top of HPAC can be achieved by optimizing thresholds for subsets of workloads. Although we did not use such an optimization for the results presented in the evaluation section, this demonstrates a rigorous and automated approach to optimization of decision and threshold sets for HPAC.

3.3 Implementation

In our implementation of HPAC, FDP, and coordinated throttling, all mechanisms are implemented using an interval-based sampling mechanism similar to that used in [27, 6]. To detect the end of an interval, a hardware counter is used to keep track of the number of cache lines evicted from the L2 cache. When the counter exceeds the empirically determined threshold of 8192 evicted lines, an interval ends and the counters gathering feedback information are updated using the following equation:

$$Count = 1/2\ CountAtStartOfInt.\ +\ 1/2\ CountDuringInt.$$

HPAC's global control mechanism maintains counters for keeping track of the BWC_i, BWN_i and POL_i at each core i as defined in Section 3.2.1. ACC_i is calculated by maintaining two counters to keep track of the number of useful prefetches for core i (*used-total*$_i$) and the total number of prefetches of that core (*pref-total*$_i$). The update of these counters is similar to that proposed for FDP. ACC_i is obtained by taking the ratio of *used-total*$_i$ to *pref-total*$_i$ at the end of every interval. BWC_i and BWN_i are maintained by simply incrementing their values at the memory controller every DRAM cycle based on the state of the requests in that cycle (see the example in Section 3.2.1).

To calculate POL_i, we need to track the number of last-level cache demand misses core i's prefetches cause for all other cores. We use a Bloom filter [2] for each core i to approximate this count. Each filter entry consists of a *pollution bit* and a *processor id*. When a prefetch from core i replaces another core j's demand line, core i's filter is accessed using the evicted line's address, the corresponding pollution bit is set in the filter, and the corresponding processor id entry is set to j. When memory finishes servicing a prefetch request from core j, the Bloom filters of all cores are accessed by the address of the fetched line and the pollution bit of that entry is reset if the processor id of the corresponding entry is equal to j. When a demand request from core j misses the last level cache, the filters of all cores are accessed using the address of that demand request. If the corresponding bit of core i's Bloom filter is set and the processor id of the entry is equal to j, the filter predicts that this line was evicted previously due to a prefetch from core i and the miss could have been avoided had the prefetch that evicted the requested line not been inserted into the cache. Hence, POL_i is incremented and the pollution bit is reset. The interval-based nature of our technique puts the communication of information needed to update pollution filters and feedback counters off the critical path since all such communication only needs to complete before the end of the current interval.

4. METHODOLOGY

Processor Model and Workloads We use a cycle accurate x86 CMP simulator for our evaluation. We faithfully model all port contention, queuing effects, bank conflicts, and other DDR3 DRAM system constraints in the memory subsystem. Table 2 shows the baseline configuration of each core and the shared resource configuration for the 4 and 8-core CMP systems we use.

Execution Core	Out of order, 15 stages
	Decode/retire up to 4 instructions
	Issue/execute up to 8 micro instructions;
	256-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 size;
	L1 D-cache: 32KB, 4-way, 2-cycle, 64B line size;
	Shared unified L2: 2MB (4MB for 8-core), 16-way (32-way for 8-core), 16-bank, 15-cycle (20-cycle for 8-core), 1 port, 64B line size;

Prefetcher	Stream prefetcher with 32 streams, prefetch degree of 4, and prefetch distance of 64 [28, 27]
DRAM controller	On-chip, demand-first [13] Parallelism-Aware Batch Scheduling policy [21] 128 L2 MSHR (256 for 8-core) and memory request buffer; Two memory channels for 8-core;
DRAM and Bus	667MHz DRAM bus cycle, Double Data Rate (DDR3 1333MHz) [17], 8B-wide data bus Latency: 15ns per command (^tRP, ^tRCD, CL); 8 DRAM banks, 16KB row buffer per bank;

Table 2: Baseline system configuration

We use the SPEC CPU 2000/2006 benchmarks for our experimental evaluation. Each benchmark was compiled using ICC (Intel C Compiler) or IFORT (Intel Fortran Compiler) with the -O3 option. We ran each benchmark with the reference input set for 200 million x86 instructions selected by Pinpoints [24] as a representative portion of each benchmark.

We classify benchmarks into *memory intensive/non-intensive*, *with/without cache locality in data accesses*, and *prefetch sensitive* for purposes of analysis in our evaluation. We refer to a benchmark as memory intensive if its L2 Cache Miss per 1K Instructions (MPKI) is greater than one. We say a benchmark has cache locality if its number of L2 cache hits per 1K instructions is greater than five, and we say it is prefetch sensitive if the performance delta obtained with an aggressive prefetcher is greater than 10% compared to no prefetching. These classifications are based on measurements made when each benchmark was run alone on the 4-core system. We show the characteristics of some (due to space limitations) of the benchmarks that appear in the evaluated workloads in Table 3.

Workload Selection We used 32 four-application and 32 eight-application multi-programmed workloads for our 4-core and 8-core evaluations. These workloads were randomly selected from all possible 4-core and 8-core workloads with the one condition that the evaluated workloads be relevant to the proposed techniques: each application in each workload is either memory intensive, prefetch sensitive, or has cache locality.

	No prefetcher			With Stream Prefetcher			
Benchmark	IPC	MPKI	Traffic	IPC	MPKI	Traffic	ACC (%)
bzip2_00	1.27	0.39	0.08	1.37	0.11	0.08	95.4
swim_00	0.36	23.10	4.62	0.70	3.43	4.62	99.9
facerec_00	1.35	2.72	0.56	1.46	1.16	0.87	60.0
parser_00	1.06	0.63	0.13	1.17	0.11	0.16	83.1
apsi_00	1.75	0.85	0.17	1.87	0.39	0.17	99.3
perlbmk_00	1.85	0.04	0.01	1.86	0.02	0.02	28.7
xalancbmk_06	0.93	0.82	0.18	0.78	1.48	0.87	8.2
libquantum_06	0.39	13.51	2.70	0.41	2.62	2.70	99.9
omnetpp_06	0.41	8.68	1.74	0.43	8.43	5.35	11.5
GemsFDTD_06	0.46	15.35	3.03	0.78	1.64	3.33	90.9
lbm_06	0.31	20.16	4.03	0.50	3.92	3.92	93.8
bwaves_06	0.58	18.7	3.74	1.02	0.57	3.74	99.8

Table 3: Characteristics of 12 SPEC 2000/2006 benchmarks with/without prefetching: IPC, MPKI, Bus Traffic (M cache lines), and ACC

Prefetcher Aggressiveness Levels and Thresholds for Evaluation Table 4 shows the values we use for determining the aggressiveness of the stream prefetcher in our evaluations. The aggressiveness of the GHB [22] prefetcher is determined by its *prefetch degree*. We use five values for GHB's prefetch degree (2, 4, 8, 12, 16). Throttling a prefetcher up/down corresponds to increasing/decreasing its aggressiveness by one level.

Threshold values for FDP [27] and coordinated throttling [6] are empirically determined for our system configuration. We use the threshold values shown in Table 5 for HPAC. We determined these threshold values empirically, but due to the large design space, we did not tune the values. Unless otherwise stated, we use FDP as the local control mechanism in our evaluations.

Aggressiveness Level	Stream Prefetcher *Distance*	Stream Prefetcher *Degree*
Very Conservative	4	1
Conservative	8	1
Moderate	16	2
Aggressive	32	4
Very Aggressive	64	4

Table 4: Pref. configurations

ACC	BWC	POL	$BWNO$
0.6	50k	90	75k

Table 5: HPAC threshold values

Metrics To measure CMP system performance, we use *Individual Speedup (IS)*, *Harmonic mean of Speedups (Hspeedup or HS)* [16], and *Weighted Speedup (Wspeedup or WS)* [26]. *IS* is the ratio of an application's performance when it is run together with other applications on different cores of a CMP to its performance when it runs alone on one core in the CMP system (other cores are idle). This metric reflects the change in performance of an application that results from running concurrently with other applications in the CMP system. Recent research [7] on system-level performance metrics for multi-programmed workloads shows that *HS* is the reciprocal of the *average turn-around time* and is the primary user-oriented system performance metric [7]. *WS* is equivalent to *system throughput* which accounts for the number of programs completed per unit of time. We show both metrics throughout our evaluation. To demonstrate that the performance gains of our techniques are not due to unfair treatment of applications, we also report *Unfairness*, as defined in [20]. Unfairness is defined as the ratio between the maximum individual speedup and minimum individual speedup among all co-executed applications.

The equations below provide the definitions of these metrics. In these equations, 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 with the prefetcher enabled (other cores are idle). $IPC^{together}$ is the IPC measured when an application runs on one core while other applications are running on the other cores.

$$IS_i = \frac{IPC_i^{together}}{IPC_i^{alone}}, \qquad WS = \sum_{i=0}^{N-1} \frac{IPC_i^{together}}{IPC_i^{alone}}$$

$$HS = \frac{N}{\sum_{i=0}^{N-1} \frac{IPC_i^{alone}}{IPC_i^{together}}}, \; Unfairness = \frac{MAX\{IS_0, IS_1, ..., IS_{N-1}\}}{MIN\{IS_0, IS_1, ..., IS_{N-1}\}}$$

5. EXPERIMENTAL EVALUATION

We evaluate HPAC on both 4-core and 8-core systems. We find the improvements provided by our technique increases as the number of cores in a CMP increases. We present both sets of results, but most of the analysis is done on the 4-core system to ease understanding.

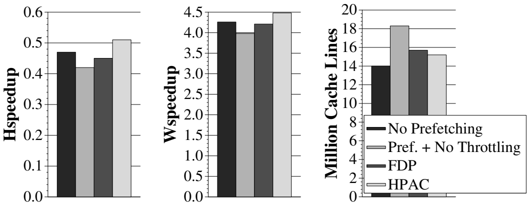
5.1 8-core System Results

Figure 5 shows system performance and bus traffic averaged across 32 workloads evaluated on the 8-core system. HPAC provides the highest system performance among all examined techniques, and is the only technique employing prefetching that improves average system performance over no prefetching. It also consumes the least bus traffic among schemes that employ prefetching. Several key observations are in order:

1. Employing aggressive prefetching with no throttling performs worse than no prefetching at all: harmonic speedup and weighted speedup decrease by 16% and 10% respectively. We conclude that attempting to aggressively prefetch in CMPs with no throttling has significant negative effects, which makes aggressive prefetching a challenge in CMP systems.

2. FDP increases performance compared to no prefetcher throttling, but is still inferior to no prefetching. FDP's performance is 4.8%/1.2% (HS/WS) lower than no prefetching while its bus traffic is 12.8% higher. We conclude that inter-core prefetcher interference, which is left unmanaged by even a state-of-the-art local-only prefetch control scheme, can cause prefetching to be detrimental to system performance in CMPs.

3. HPAC improves performance by 8.5%/5.3% (HS/WS) compared to no prefetching, at the cost of only 8.9% higher bus traffic. In addition, HPAC increases performance by 23% and 14% (HS), and consumes 17% and 3.2% less memory bandwidth compared to no throttling and FDP respectively, as summarized in Table 6. HPAC enables prefetching to become effective in CMPs by controlling and reducing prefetcher-caused interference. Among the schemes where prefetching is enabled, HPAC is the most bandwidth-efficient. We conclude that with HPAC, prefetching can significantly improve system performance of CMP systems without large increases in bus traffic.



(a) Hspeedup (b) Wspeedup (c) Bus Traffic

Figure 5: HPAC performance on 8-core system (all 32 workloads)

As an example of how HPAC performs compared to other schemes on different workloads, Figure 6 shows the performance improvement (in terms of harmonic speedup) of no prefetching, FDP, and HPAC normalized to that of prefetching with no throttling across eight of the 32 evaluated workloads. We do not present a per-workload analysis of these workloads due to space constraints but we do present a thorough analysis of workload characteristics for 4-core workloads in Section 5.2.

	HS	WS	Bus Traffic
HPAC Δ over No Prefetching	8.5%	5.3%	8.9%
HPAC Δ over No Throttling	23%	12.8%	-17%
HPAC Δ over FDP	14%	6.6%	-3.2%

Table 6: Summary of average results on the 8-core system

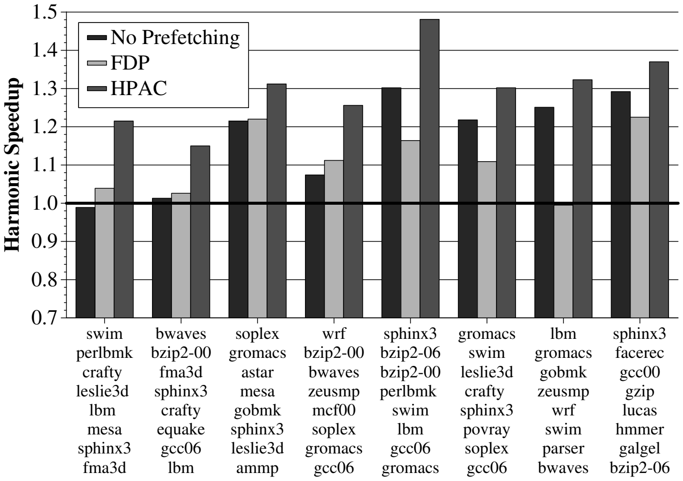


Figure 6: Hspeedup of 8 of the 32 workloads
(normalized to "no throttling")

5.2 4-core System Results

We first present overall performance results for the 32 workloads evaluated on the 4-core system, and analyze the workloads' characteristics. We then discuss a case study in detail to provide insight into the behavior of the scheme.

5.2.1 Overall Performance

Table 7 summarizes our overall performance results for the 4-core system. As observed with the 8-core workloads in Section 5.1, HPAC provides the highest system performance among all examined techniques. It also generates the least bus traffic among schemes that employ prefetching.

	HS	WS	Bus Traffic
HPAC Δ over No Prefetching	8.9%	5.3%	8.9%
HPAC Δ over No Throttling	15%	8.4%	-14%
HPAC Δ over FDP	10.7%	4.7%	-3.2%

Table 7: Summary of average results on the 4-core system

Workload Analysis: Figure 7 shows the performance improvement (in terms of harmonic speedup) of no prefetching, FDP, and HPAC normalized to that of prefetching with no throttling across 14 of the 32 evaluated workloads. Four distinct classes of workloads can be identified from this figure.

Class 1: Prefetcher-caused inter-core interference does not allow significant gains with no throttling or FDP. In fact, in the leftmost two cases, FDP degrades performance slightly compared to no throttling because it increases prefetchers' interference in the shared resources (as discussed in detail in the case study presented in Section 5.2.2). HPAC controls this interference and enables much higher system performance improvement than what is possible without it.

Class 2: Significant performance can be obtained with FDP and sometimes with no throttling since prefetcher-caused inter-core interference is tolerable. HPAC performs practically at least as well as these previous mechanisms.

Class 3: Intense prefetcher-caused inter-core interference makes prefetching significantly harmful with no throttling or FDP. FDP can slightly reduce this interference compared to

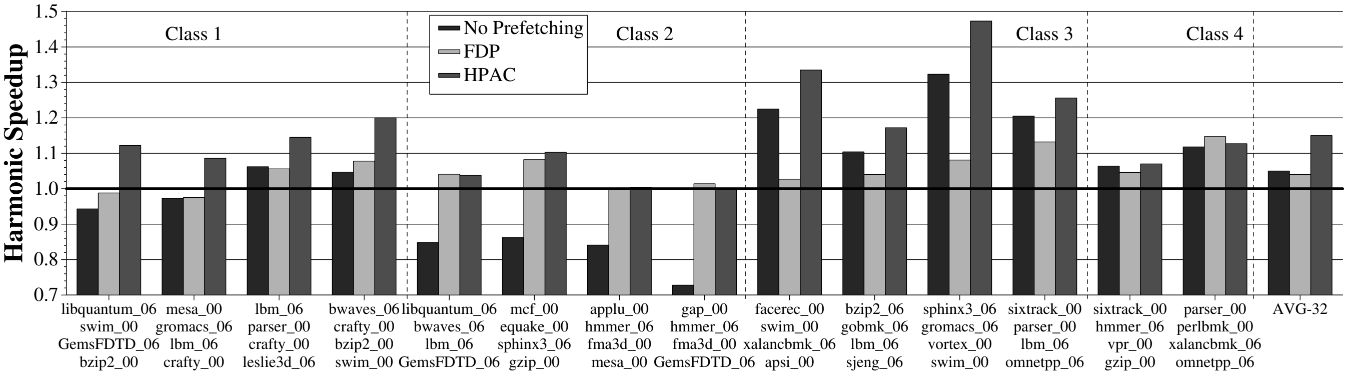
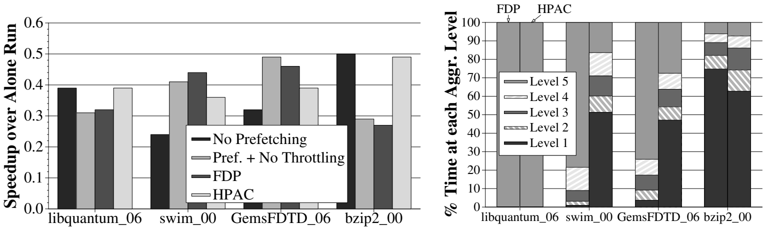
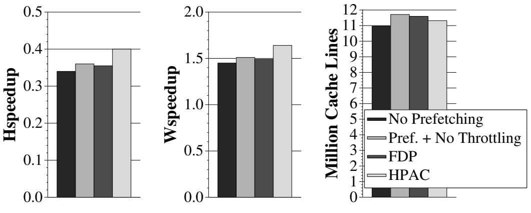


Figure 7: Hspeedup of 14 of the 32 workloads (normalized to "no throttling")



(a) Individual Speedup (b) Prefetch Aggr. Levels

Figure 8: Case Study: individual application behavior



(a) Hspeedup (b) Wspeedup (c) Bus Traffic

Figure 9: Case Study: system behavior

no throttling by making prefetchers independently more accurate, but still degrades performance significantly compared to no prefetching. The existence of such workloads makes prefetching without control of prefetcher-caused inter-core interference very unattractive in CMPs. However, HPAC enables prefetching to significantly improve performance over no prefetching.
Class 4: Small prefetcher-caused inter-core interference can be controlled by FDP. Potential system performance to be gained by prefetching is small compared to other classes. Small performance degradations of no throttling can be eliminated using FDP or HPAC, which perform similarly.

We conclude that HPAC is effective for a wide variety of workloads. In many workloads where there is significant prefetcher-caused inter-core interference (classes 1 and 3), HPAC is the only technique that enables prefetching to improve performance significantly over no prefetching. When prefetcher-caused inter-core interference is not significant (class 2), HPAC retains significant performance over no prefetching. Hence, HPAC makes prefetching effective and robust in multi-core systems.

5.2.2 Case Study

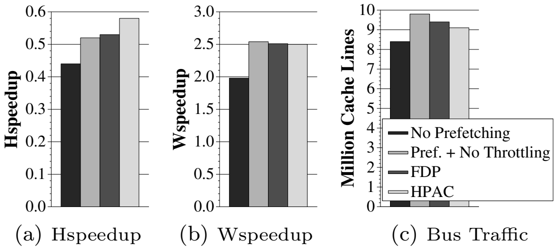
This case study is an example of a scenario where prefetcher-caused inter-core interference that hampers system performance can be observed in both shared bandwidth and shared cache space. It provides insight into why controlling aggressiveness of a CMP's prefetchers based on local-only feedback of each core is ineffective.

We examine a scenario where a combination of three memory-intensive applications (*libquantum, swim, GemsFDTD*) are run together with one memory non-intensive application that has high data cache locality (*bzip2*). Figures 8 and 9 show individual benchmark performance and overall system performance, respectively. Several observations are in order:

First, employing aggressive prefetching on all cores improves performance by 6.0%/3.7% (HS/WS) compared to no prefetching. However, the effect of prefetching on individual benchmarks is mixed: even though two applications' (*swim* and *GemsFDTD*) performance significantly improves, that of two others (*libquantum* and *bzip2*) significantly degrades. Although *libquantum*'s prefetches are very accurate, they, along with *libquantum*'s demands, are delayed by *swim*'s and *GemsFDTD*'s prefetches in the memory controller. Since previous works [13, 6] analyzed the effects of enabling prefetching in multi-core systems, we focus our analysis on the differences between prefetching without throttling, local-only throttling, and HPAC.

Second, using FDP to reduce the negative effects of prefetching actually degrades system performance by 1.2%/1% (HS/WS) compared to no throttling. To provide insight, Figure 8(b) shows the percentage of total execution time each application's prefetcher spends in different aggressiveness levels. With FDP, since the feedback indicates high accuracy for prefetchers of *libquantum, swim* and *GemsFDTD* (respectively at accuracies of 99.9%, 99.9%, 92%), their prefetchers are kept very aggressive. This causes significant memory bandwidth interference between these three applications, which causes *libquantum*'s demand and prefetch requests to be delayed by the aggressive *swim* and *GemsFDTD* prefetch requests. On the other hand, *bzip2*'s demand-fetched cache blocks get thrashed due to the very large number of *swim*'s and *GemsFDTD*'s prefetches: *bzip2*'s L2 demand MPKI increases by 26% from 2.1 to 2.7. *bzip2*'s prefetcher performance is also affected negatively as its useful prefetches are evicted from the cache before being used and therefore reduced by 40%. This prompts FDP to reduce the aggressiveness of *bzip2*'s prefetcher as a result of detected *local* low accuracy, which in turn causes a loss of potential performance improvement for *bzip2* from prefetching. As a result, FDP does not help *libquantum*'s performance and degrades *bzip2*'s performance, resulting in overall system performance degradation compared to no throttling.

Third, using HPAC increases system performance significantly by 12.2%/8.7% (HS/WS) while reducing bus traffic by 3.5% compared to no throttling. Hence, HPAC makes aggressive prefetching significantly beneficial to the entire system: it increases performance by 19%/12.7% (HS/WS) compared to a system with no prefetching. The main reason for the performance benefits of HPAC over FDP is twofold: 1) by tracking prefetcher-caused interference in the shared cache, HPAC recognizes that aggressive (yet accurate) prefetches of *swim* and *GemsFDTD* destroy the cache locality of *bzip2* and throttles those applications' prefetchers, thereby significantly improving *bzip2*'s locality and performance, 2) by tracking the bandwidth need and bandwidth consumption of cores in the DRAM



(a) Hspeedup (b) Wspeedup (c) Bus Traffic

Figure 10: Performance of HPAC on system using PADC

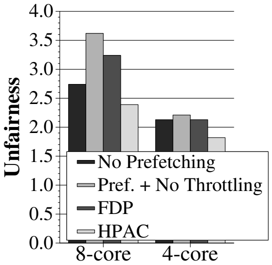
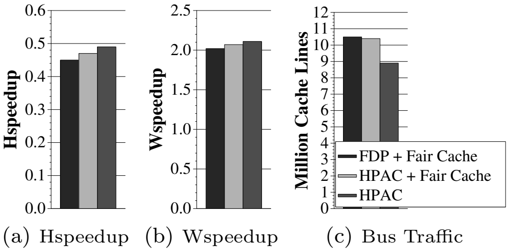


Figure 11: Unfairness in 8- and 4-core systems



(a) Hspeedup (b) Wspeedup (c) Bus Traffic

Figure 12: Comparison to combination of fair cache + fair memory scheduling + FDP

system, HPAC recognizes that *swim*'s and *GemsFDTD*'s aggressive prefetches delay service of *libquantum*'s demands and prefetches, and therefore throttles down these two prefetchers. Doing so significantly improves *libquantum*'s performance. HPAC improves the performance of all applications compared to no prefetching, except for *bzip2*, which still incurs a slight (1.5%) performance loss. Finally, HPAC reduces memory bus traffic compared to both FDP and no throttling because: 1) it eliminates many unnecessary demand requests that need to be re-fetched from memory by reducing the pollution *bzip2* experiences in the shared cache: *bzip2's* bandwidth demand reduces by 33% with HPAC compared to FDP, 2) it eliminates some useless (or marginally useful) prefetch requests due to *GemsFDTD's* very aggressive prefetcher: we found that in total, HPAC reduces the number of useless prefetch requests by 14.6% compared to FDP.

Table 8 and Figure 8(b) provide more insight into the behavior and benefits of HPAC by showing the most common global control cases (from Table 1) for each application and the percentage of time each prefetcher spends at different levels of aggressiveness respectively. Note that Case 14, which indicates extreme prefetcher interference is *swim*'s and *GemsFDTD*'s most frequent case. As a result, HPAC throttles down their prefetchers to reduce the interference they cause in shared resources. Figure 8(b) shows that FDP keeps these two applications' prefetchers at the highest aggressiveness for more than 70% of their execution time, which degrades system performance, because FDP cannot detect the inter-core interference caused by the two prefetchers. In contrast, with HPAC, the two prefetchers spend approximately 50% of their execution time in the lowest aggressiveness level, thereby reducing inter-core interference and improving system performance.

Application	Most Frequent Case #	2nd Most Frequent Case #	3rd Most Frequent Case #
libquantum	Case 6 (89%)	Case 13 (7%)	Case 7 (2%)
swim	Case 14 (65%)	Case 7 (23%)	Case 6 (6%)
GemsFDTD	Case 14 (55%)	Case 7 (24%)	Case 6 (8%)
bzip2	Case 10 (39%)	Case 3 (39%)	Case 6 (15%)

Table 8: Most frequently exercised cases for HPAC in case study I

We conclude that HPAC can effectively control and reduce the shared resource interference caused by the prefetchers of multiple memory- and prefetch-intensive applications both among themselves and against a simultaneously running memory non-intensive application, thereby resulting in significantly higher system performance than what is possible without it.

5.3 HPAC Performance with Different DRAM Scheduling Policies

We evaluate the performance of our proposal in a system with the recently proposed Prefetch-Aware DRAM Controller (PADC) [13]. PADC uses feedback about the accuracy of the prefetcher of each core to adaptively prioritize between prefetch requests of that prefetcher and demands in memory scheduling decisions. If the prefetcher of a core is accurate, prefetch requests from that core are treated with the same priority as demand requests. Otherwise, prefetches from that core are deprioritized below demands and prefetches from cores with high prefetch accuracy. Note that this local-only technique does not take into account inter-core interference caused by prefetchers. If the memory scheduler increases the priority of highly accurate but interfering prefetches, inter-core interference will likely increase. As a result, PADC cannot control the negative performance impact of accurate yet highly-interfering prefetchers in the memory system, which can degrade system performance.

Figure 10 shows the effect of HPAC when employed in a system with a prefetch-aware DRAM controller. HPAC increases the performance of a 4-core system that uses PADC by 12% (HS) on average while reducing bus traffic by 7%. HPAC's ability to reduce the negative interference caused by accurate prefetchers can have positive effects on PADC's options for better memory scheduling when PADC and HPAC are employed together. A reduction in interference caused by one core's very aggressive prefetcher can reduce the number of demand misses of other cores. This removes many pollution-induced misses caused by the interfering core(s) and the new miss stream observed by the prefetchers of other cores can increase their accuracy significantly. HPAC's interference reduction enables PADC's memory scheduling decisions to take advantage of these more accurate prefetches. In contrast, PADC without HPAC would have seen inaccurate prefetch requests from such cores and deprioritized them due to their low accuracy. We conclude that systems with PADC-like memory controllers can benefit significantly if their prefetchers are controlled in a coordinated manner using HPAC.

The performance and bus traffic benefits of using HPAC with an FR-FCFS [25] memory scheduling policy are similar to those presented for the PAR-BS [21] fair memory scheduler which we use as our baseline (i.e., 12.4%/6.2% HS/WS improvement over FDP). We conclude that our proposal is orthogonal to the baseline memory scheduling policy.

5.4 Effect of HPAC on Fairness

Although HPAC's objective is to "improve system performance" not to "improve fairness," it is worth noting that HPAC's performance improvement does not come at the expense of fair treatment of all applications. We have evaluated HPAC's impact on performance unfairness [20] as defined in Section 4. Figure 11 shows that HPAC actually reduces unfairness in the system compared to all other techniques in both 4-core and 8-core systems. We found that this is because HPAC significantly reduces the interference caused by applications that generate a very large number of prefetches on other less memory-intensive applications. This interference unfairly slows down the latter type of applications in the baseline since there is no mechanism that controls such interference.

We note that HPAC is orthogonal to techniques that provide fairness in shared resources [23, 10, 21]. As such, HPAC can be combined with techniques that are designed to provide fairness in shared multi-core resources. Note that we use Parallelism-Aware Batch Scheduling [21] as a fair memory scheduler in the *baseline* for all our evaluations. Figure 12 shows system performance and bus traffic of a 4-core system that uses a fair cache [23], a fair memory scheduler [21] and a state-of-the-art local-only prefetcher throttling mechanism (FDP) compared to 1) the combination of HPAC and a fair cache, and 2) HPAC by itself. Two observations are in order: First, using HPAC improves the performance of a system employing a fair cache. However, the improvement in performance is less than that obtained by HPAC alone. The reason is that constraining each core to a certain number of ways in each cache set as done in [23] reduces HPAC's flexibility. HPAC can throttle down a prefetcher that is causing large inter-core pollution to reduce such interference without the constraints of a fair cache [23]. Therefore HPAC can make more efficient use of cache space and perform better alone. Second, HPAC outperforms the combination of a fair cache, a fair memory scheduler, and FDP, by 10.2% (HS) and 4.7% (WS) while consuming 15% less bus traffic. We conclude that 1) our contribution is orthogonal to techniques that provide fairness in shared resources, and 2) the benefits of adjusting the aggressiveness of multiple prefetchers in a coordinated fashion (as done by HPAC) cannot be obtained by combining FDP, a fair cache, and a fair memory controller.

5.5 Multiple Types of Prefetchers per Core

Recent research suggests that by using "coordinated throttling" of multiple prefetchers of different types, hybrid prefetching systems can be useful [6]. Some current processors already employ more than one type of prefetcher on each core of a CMP [30]. We evaluate the effectiveness of our proposal on a 4-core system with two types of prefetcher per core and also with two different state-of-the-art local control policies as the local control for HPAC: FDP [27] and coordinated throttling [6]. Tables 9 and 10 show that HPAC is effective: 1) when multiple prefetchers of different types are employed within each core and 2) regardless of the local throttling policy used for prefetchers of each core. In all comparisons HPAC is the best performing of all schemes and produces the least bus traffic compared to any configuration with prefetching turned on.

	HS	WS	Bus Traffic
Δ over No Prefetching	7.9 %	5.1 %	10.7 %
Δ over Prefetching w. no Throttling	15.6 %	6.7 %	-13.9 %
Δ over FDP	10.6 %	3.2 %	-3 %

Table 9: Stream and GHB with HPAC (local policy: FDP)

	HS	WS	Bus Traffic
Δ over No Prefetching	6.3 %	4.0 %	12.2 %
Δ over Prefetching w. no Throttling	14.6 %	6.3 %	-12.7 %
Δ over coordinated throttling	12.2 %	4.5 %	-6.3 %

Table 10: Stream and GHB with HPAC
(local policy: coordinated throttling)

5.6 Sensitivity to System Parameters

We evaluate the sensitivity of our technique to three major memory system parameters: L2 cache size, memory latency and number of memory banks. Table 11 shows the change in system performance (HS) and bus traffic provided by HPAC over FDP for each configuration. For these experiments we did not tune HPAC's parameters; doing so will likely increase HPAC's benefits even more. We conclude that our technique is effective for a wide variety of system parameters.

5.7 Hardware Cost

Table 12 shows HPAC's required storage. The additional storage is 15.14KB (for a 4-core system), most of which is already required to implement FDP. This storage corresponds to 0.739% of the 2MB L2 baseline cache. The new global control structures require only 1.55KB of storage (for a 4-core system) on top of FDP. HPAC does not require any structures or logic that are on the critical path of execution.

L2 Cache Size					
1 MB		2 MB		4 MB	
Δ HS	Δ Bus Traffic	Δ HS	Δ Bus Traffic	Δ HS	Δ Bus Traffic
19.5%	-4%	10.7%	-3.2%	9.6%	-2.5%

Memory Latency - Latency per command (^tRP, ^tRCD, CL)					
13ns		15ns		17ns	
Δ HS	Δ Bus Traffic	Δ HS	Δ Bus Traffic	Δ HS	Δ Bus Traffic
15 %	-3%	10.7%	-3.2%	6%	-3.4%

Number of Memory Banks					
8 banks		16 banks		32 banks	
Δ HS	Δ Bus Traffic	Δ HS	Δ Bus Traffic	Δ HS	Δ Bus Traffic
10.7%	-3.2%	12%	-1.5%	9%	-1%

Table 11: Effect of our proposal on Hspeedup (HS) and bus traffic with different system parameters on a 4-core system

Global Control	Closed form for N cores (bits)	N=4(bits)
Counters for global feedback	7 counters/core×N×16 bits/counter	448
Interference Pol. Filter per core	1024 entries × N × (pol. bit+(log N) bit proc. id)/entry	12,288
Local Control - FDP		
Proc. id for each L2 tag store entry	16384 blocks/Megabyte × S_{cache} × (log N) bit/block	65,536
Pref. bit for each L2 tag store entry	16384 blocks/Megabyte × S_{cache} × 1 bit/block	32,768
Pol. Filter for intra-core prefetch interference	1024 entries× N × (pol. bit+(log N) bit proc. id)/entry	12,288
Counters for local feedback	(8 counters/core×N + 3 counters) ×16 bits/counter	560
Pref. bit per MSHR entry	32 entries/core × N × 1 bit/entry	128
Total storage	Sum of the above	15.14 KB

Table 12: Hardware cost of HPAC - Including both local and global throttling structures on an N-core CMP with S_{cache} MB L2 cache

6. RELATED WORK

To our knowledge, this paper provides the first comprehensive solution for coordinated control of multiple prefetchers of multiple cores in a CMP environment. The major contribution of our proposal is a hierarchical prefetcher aggressiveness control technique that incorporates system-wide inter-core prefetcher interference feedback into the decision making process of how aggressive the prefetcher(s) of each core should be. In this section, we briefly discuss and compare to related work in prefetcher control, useless prefetch elimination, and cache pollution reduction.

6.1 Per-Core Prefetcher Control

We have already shown that HPAC significantly improves system performance and bandwidth efficiency over two purely-local prefetcher control techniques [27, 6] and is orthogonal to them. Other prior work proposed dynamically configuring the aggressiveness of the prefetcher of a core or turning off prefetchers based on their accuracy [22, 8]. Nesbit et al. [22] propose per program-phase dynamic configuration of the prefetch degree of a GHB prefetcher. Gendler et al. [8] propose a multiple prefetcher handling mechanism (called PAB) that turns off all prefetchers but the most accurate one based on only per-prefetcher accuracy data obtained from the last N prefetched addresses. All such techniques that use only *local* feedback information to either turn off or throttle the prefetcher(s) can significantly degrade performance because they can exacerbate interference in shared resources.

Prior work in shared memory multiprocessors Prior works on prefetching in multiprocessors [5, 29] study adaptivity and limitations of prefetching in these systems. Dahlgren et al. [5] use prefetch accuracy to decide whether to increase or decrease aggressiveness on a per-processor basis, similar to employing FDP on each core's prefetcher independently. Tullsen and Eggers [29] develop a prefetching heuristic tailored to write-shared data in multi-threaded applications. They apply a restructuring algorithm for shared data to reduce false sharing in multi-threaded applications. In contrast to these prior works, our goal is to make prefetching effective by controlling prefetch-caused *inter-application* interference. None of these prior works solve the problem we target and therefore are ineffective in reducing prefetcher-caused inter-application interference.

6.2 Eliminating Useless Prefetches

Many previous proposals address the problem of useless prefetches by proposing mechanisms to intelligently filter them out [18, 3, 15, 31, 19, 13]. HPAC is complementary to these prefetch filtering techniques because it can reduce the harmful effects of not only inaccurate prefetchers but also accurate yet interfering prefetchers, as we showed in our case study. Hence, HPAC can be used in conjunction with prefetch filtering to attain even higher performance.

Zhuang and Lee [31] propose a hardware-based prefetch filtering scheme that eliminates a prefetch request for an address if a prefetch request for the same address was useless in the past. They use a two-level branch predictor-like structure to record the usefulness of prefetches. We implemented HPAC on top of this hardware filtering scheme, and found that HPAC increases system performance by 12% while reducing bus traffic by 8.7% compared to hardware filtering alone on the evaluated 4-core workloads. We find that hardware filters and HPAC work synergistically: together, they perform better than each one alone.

6.3 Reducing Cache Pollution

Cache pollution caused by prefetches can be reduced by using separate prefetch buffers [14] instead of inserting prefetched data into the last level cache. However, doing so 1) may lead to eviction of prefetches from these buffers before they are used due to the limited storage capacity of prefetch buffers, and 2) increases design complexity of the memory system. Prior research [27] showed that in order to provide significant performance improvements, the size of the prefetch buffers needs to be very large (at least 64KB). In this work, we propose a significantly lower-cost and simpler technique that does not require separate buffer structures and data paths in the memory system to treat prefetches specially.

7. CONCLUSION

We have proposed a low-cost technique that controls the aggressiveness of multiple prefetchers of different cores in chip-multiprocessors with the goal of improving system performance and making prefetching effective. We show that adjusting prefetcher aggressiveness using state-of-the-art techniques without paying attention to prefetcher-caused inter-core interference in shared memory systems can significantly degrade system performance compared to no prefetching at all. The key idea of our solution is to take into account prefetcher-caused inter-core interference in determining the aggressiveness of each core's prefetcher. Our scheme reduces the interference due to prefetchers using a coordinated control mechanism, thereby significantly improving system performance and bandwidth-efficiency compared to the state-of-the-art prefetcher control techniques that do not take into account such interference. We conclude that our technique significantly improves the performance of prefetching and makes it effective in multi-core environments.

Acknowledgments

Many thanks to Jose Joao, Veynu Narasiman, Rustam Miftakhutdinov, Khubaib, and other HPS members and the anonymous reviewers for their comments and suggestions. We gratefully acknowledge the support of the Cockrell Foundation, Intel, and Microsoft. We also acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources.

REFERENCES

[1] J. Baer and T. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing '91*, 1991.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13, 1970.

[3] M. Charney and T. Puzak. Prefetching and memory system behavior of the SPEC95 benchmark suite. *IBM Journal of Research and Development*, 31(3):265–286, 1997.

[4] R. Cooksey et al. A stateless, content-directed data prefetching mechanism. In *ASPLOS-X*, 2002.

[5] F. Dahlgren et al. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *ICPP-22*, 1993.

[6] E. Ebrahimi et al. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA-15*, 2009.

[7] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.

[8] A. Gendler et al. A PAB-based multi-prefetcher mechanism. *Intl. Journal of Parallel Programming*, 34(2):171–188, Apr. 2006.

[9] D. E. Goldberg and J. H. Holland. Genetic algorithms and machine learning. *Journal of Machine Learning*, 3(2-3):95–99, 1988.

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

[11] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *ISCA-24*, 1997.

[12] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA-17*, 1990.

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

[14] R. L. Lee, P.-C. Yew, and D. H. Lawrie. Data prefetching in shared memory multiprocessors. In *ICPP-16*, 1987.

[15] W.-F. Lin et al. Filtering superfluous prefetches using density vectors. In *ICCD-19*, 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] T. C. Mowry et al. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS-5*, 1992.

[19] O. Mutlu et al. Using the first-level caches as filters to reduce the pollution caused by speculative memory references. *Intl. Journal of Parallel Programming*, 33(5):529–559, October 2005.

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

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

[22] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In *PACT*, 2004.

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

[24] H. Patil et al. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.

[25] S. Rixner et al. Memory access scheduling. In *ISCA-27*, 2000.

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

[27] S. Srinath et al. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA-13*, 2007.

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

[29] D. M. Tullsen and S. J. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *ISCA-20*, 1993.

[30] O. Wechsler. Inside Intel core microarchitecture. *Intel Technical White Paper*, 2006.

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