Joint Exploration of Hardware Prefetching and Bandwidth Partitioning in Chip Multiprocessors

Fang Liu
fliu3@ece.ncsu.edu

Yan Solihin
solihin@ece.ncsu.edu

Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC, USA

ABSTRACT

In this paper, we propose an analytical model-based study to investigate how hardware prefetching and memory bandwidth partitioning impact Chip Multi-Processors (CMP) system performance and how they interact. The model includes a composite prefetching metric that can help determine under which conditions prefetching can improve system performance, a bandwidth partitioning model that takes into account prefetching effects, and derives weighted speedup optimum bandwidth partition sizes for different cores. Simulation results show that selective prefetching, guided by the composite prefetching metric, coupled with dynamic bandwidth partitioning can improve system performance by 19.5% and 21.8% on a dual-core CMP and quad-core CMP respectively.

1. INTRODUCTION

Several decades of the persistent gap between microprocessor and DRAM main memory speed improvement has made it imperative for today's microprocessors to hide hundreds of processor clock cycles in memory access latency. In most high performance microprocessors today, this is achieved by employing several levels of caches augmented with *hardware prefetching* [3, 8]. Hardware prefetching works by predicting cache blocks that are likely needed by the processor in the near future, and fetching them into the cache early, allowing processor's memory accesses to hit (i.e. find data) in the cache. Since prefetching relies on prediction of future memory accesses, it is not 100% accurate in its prediction, implying that the latency hiding benefit of prefetching causes an increase in off-chip memory bandwidth usage and cache pollution.

Unfortunately, the shift to multi-core design has significantly increased the pressure on the off-chip bandwidth. Moore's Law continues to allow the doubling of transistors (cores) every two years, increasing the off-chip bandwidth traffic at Moore's Law speed. However, the availability of off-chip bandwidth is only projected to grow at 15% annually, due to the limitations in pin density and power consumption [9]. This leads to a problem called the *bandwidth wall*, where system performance is increasingly limited by the availability of off-chip bandwidth [2]. Hence, the demand for efficient use of off-chip bandwidth is tremendous and increasing.

Two methods have been recently proposed to improve the efficiency of off-chip bandwidth usage. One method is to improve prefetching policies. Some studies proposed throttling or eliminating the useless/bad prefetches from consuming bandwidth [11, 18], and tweaking the memory scheduling policy to prioritize demand and profitable prefetch requests [4]. Another method is to partition the off-chip bandwidth among cores [5, 7, 14, 13, 10, 17], by choosing partition sizes to optimize system throughput or fairness. However, these studies suffer from several drawbacks. First, the studies address one technique but ignore the other: prefetching studies do not include bandwidth partitioning, whereas bandwidth partitioning studies assume systems that have no prefetching. As a result, the significant interaction between them was missed, and the opportunity for these techniques to work in synergy was left unexplored. Second, the studies were performed in an ad-hoc manner, yielding performance improvement but missing important insights.

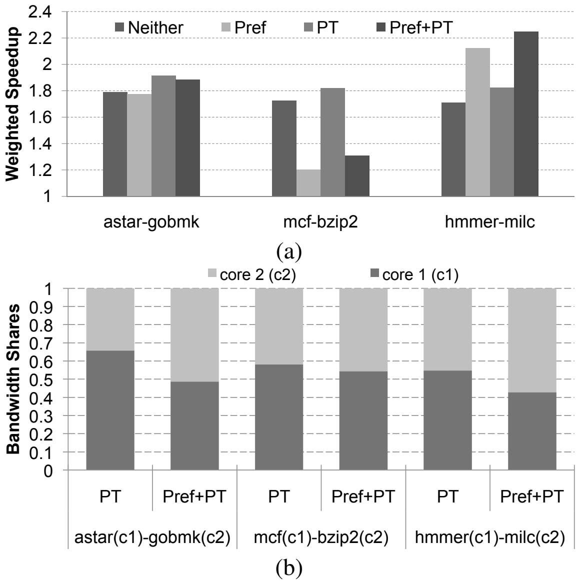


Figure 1: Weighted speedup (a) and optimum bandwidth allocation (b) for co-schedules running on a dual-core CMP.

To demonstrate the need for exploring prefetching and bandwidth partitioning jointly, Figure 1(a) shows how prefetching and bandwidth partitioning can affect each other. In the figure, system performance measured as weighted speedup [1] (referring to Equation 1) of three pairs of SPEC2006 [19] benchmarks running as a co-schedule on a dual-core CMP system is shown with four configurations: base system with no prefetching or bandwidth partitioning (Neither), hardware prefetching only (Pref), bandwidth partitioning only (PT), and both prefetching and bandwidth partitioning (Pref+PT) [1]. The three co-schedules highlight different interaction cases. For *hmmer-milc*, prefetching improves performance and bandwidth partitioning improves it further due to offsetting the effect of the prefetcher's increase in off-chip bandwidth usage. For *astar-gobmk* and *mcf-bzip2*, prefetching hurts performance due to high off-chip bandwidth consumption. For *mcf-bzip2*, applying bandwidth partitioning is not the right solution, since it cannot recover the lost performance due to prefetching (i.e. Pref+PT $<$ Nei-

[1]A stream prefetcher [3, 8] is used. Optimum bandwidth partitioning scheme from [7] is used. More details can be found in Section 4.1

ther). Such a conclusion cannot be derived in prior studies, when only bandwidth partitioning was studied [5, 7, 14, 13, 10, 17], or when prefetchers were always turned on [4]. The figure points out the need to understand when the prefetcher of each core should be turned on or off, and how bandwidth partitioning should be implemented in order to optimize system performance.

Performing the studies in an ad-hoc manner often misses important insights. For example, it has been assumed that a core that enjoys useful prefetches should be rewarded with a higher bandwidth allocation, whereas a core for which prefetching is less useful should be constrained with a lower bandwidth allocation [4]. However, our experiments show the opposite. Maximum weighted speedup is achieved when a core with highly useful prefetching is given less bandwidth allocation, and the resulting excess bandwidth is given to cores with less useful prefetching. Figure 1(b) shows the bandwidth allocation for each core that maximizes the system throughput. The application that has more useful prefetching (higher prefetching coverage and accuracy) runs on Core 1. Comparing the two bars for each co-schedule, it is clear that in order to maximize system throughput, it is the applications that show less useful prefetching should receive higher bandwidth allocations.

The goal of this study is to *understand what factors contribute to the impact of prefetching and bandwidth partitioning on CMP system performance, and how they interact*. We propose an analytical model-based study, based on the Cycle Per Instruction (CPI) model [6] and queuing theory [12], taking various system parameters (CPU frequency, cache block size, and available bandwidth), and application cache behavior metrics (miss frequency, prefetching frequency, prefetching coverage and accuracy, and various CPIs) as input. Studying the model, coupled with empirical evaluation, we arrive at several interesting findings, among them are:

- Deploying prefetching makes the available bandwidth scarcer and therefore increases the effectiveness of bandwidth partitioning in improving system performance.
- The decision of turning prefetchers on or off should be made prior to employing bandwidth partitioning, because the performance loss due to prefetching in bandwidth-constrained systems cannot be fully repaired by bandwidth partitioning. We discover a new metric to decide whether to turn on or off the prefetcher for each core.
- The theoretical optimum bandwidth allocations can be derived, and a simplified version that is implementable in hardware can approximate it quite well.
- The conventional wisdom that rewards a core that has more useful prefetching with a larger bandwidth allocation is incorrect when prefetchers of all cores are turned on. Instead, its bandwidth allocation should be slightly more constrained. Our model explains why this is so.

The findings in our study carry significant implications for CMP design and performance optimization. As the number of cores on a chip continues to double every 18-24 months, versus 10-15% ITRS' projected growth in off-chip bandwidth [9], the off-chip bandwidth available per core will become increasingly scarcer, increasing the demand for better off-chip bandwidth management.

The remainder is organized as follows: Section 2 reviews the related work, Section 3 shows the construction of the analytical model, Section 4 validates the model-driven findings through simulation based evaluation, and Section 5 concludes this paper.

2. RELATED WORK

Bandwidth partitioning. Researchers have proposed various bandwidth partitioning techniques to mitigate CMP memory bandwidth contention-related problems. The implication is to prioritize memory requests from different cores based on heuristics, in order to meet the objectives of Quality of Service (QoS) [10], fairness [14, 13], or throughput [5]. Liu et al. [7] investigated the impact of bandwidth partitioning on CMP system performance and its interaction with shared cache partitioning. Srikantaiah and Kandemir [17] explored a symbiotic partitioning scheme on the shared cache and off-chip bandwidth via empirical models. None of those bandwidth partitioning techniques took prefetching into account.

Prefetching. Hardware prefetching is widely implemented in commercial microprocessors [3, 8] to hide the long memory access latency. Due to the imperfect accuracy, prefetching incurs cache pollution and increases off-chip bandwidth consumption. Prior studies have looked into how to make prefetching more effective in CMP. Techniques include throttling/filtering useless prefetching requests [11, 18] to reduce their extra bandwidth consumption. All of the above studies ignore bandwidth contention that arises from demand and prefetch requests coming from different cores.

Prefetching and Bandwidth Partitioning. Only recently, prefetching and bandwidth partitioning in CMP were studied together as inter-related problems. Ebrahimi et al. [4] proposed to partition bandwidth usage in the presence of prefetching in CMP in order to reduce the inter-core interference. While their results show improved system performance, many questions are left unanswered. For example, what fundamental factors affect the effectiveness of prefetching and bandwidth partitioning? How do they interact with each other? Does combining prefetching and bandwidth partitioning always achieve better performance than using only one of them? What bandwidth shares can produce optimum system performance? The goal of this paper is to find out these answers, that are critical for designing a good policy for optimizing system performance. Due to the lack of understanding on these issues, the study [4] made a critical error of always keeping the prefetching engines of all cores on regardless of the situations. Figure 1 shows that in some cases, it is better to turn the prefetchers completely off.

3. ANALYTICAL MODELING

3.1 Assumptions and Model Parameters

Assumptions. This study assumes a CMP with homogeneous cores, where each core has private L1 (instruction and data) and L2 (unified) caches. Each L2 cache has a hardware prefetcher that can be turned on or off. The off-chip bandwidth is shared by all cores through a single queue interface, where all off-chip memory requests are served in First-Ready First-Come-First-Serve (FR-FCFS) policy [16]. Our study focuses on multi-programmed workloads, hence we assume that single-threaded applications run on different cores and do not share data. This means the coherence traffic is ignored. The write backs are not modeled in the system, because they are much fewer than read/write misses and not on the critical path of performance.

We define bandwidth partitioning as allocating fractions of off-chip bandwidth to different cores and enforcing these fractions as per-core quota. Thus, bandwidth partitioning reduces inter-core interference, without changing the underlying memory access scheduling policy. The bandwidth partitioning is implemented using token bucket algorithm [7], with the goal to optimize the system throughput expressed as weighted speedup, which is the sum of individual Instruction Per Cycle (IPC) speedup from each core [1] [2]:

$$WS = \sum_{i=1}^{N} \frac{IPC_i}{IPC_{alone,i}} = \sum_{i=1}^{N} \frac{CPI_{alone,i}}{CPI_i} \qquad (1)$$

[2] IPC_i and $IPC_{alone,i}$ are IPC of thread i when it runs in a co-schedule and when it runs alone in the system, respectively. Weighted speedup is a widely used metric, and includes some fairness, as speeding up one core at the expense of others.

Model Parameters. The input to the model includes system parameters as well as per-thread parameters listed in Table 1.

Table 1: Input and parameters used in our model.

System parameters	
f	CPU clock frequency (Hz)
K	Cache block size (Bytes)
B	Peak off-chip memory bandwidth (Bytes/sec)
N	Number of cores in the CMP

Thread-specific parameters	
$CPI_{L2\infty,i}$	Thread i's CPI assuming infinite L2 cache size
$CPI_{alone,i}$	Thread i's CPI when it runs alone in the CMP
M_i	Thread i's L2 cache miss rate without prefetching
A_i	Thread i's L2 cache access frequency (#accesses/sec)
$M_{p,i}$	Thread i's L2 cache miss rate after prefetching
P_i	Thread i's L2 prefetching rate (#prefetches/#accesses)
c_i	Thread i's prefetching coverage
a_i	Thread i's prefetching accuracy
$T_{m,i}$	Thread i's average memory access latency (#cycles)
β_i	Fraction of bandwidth assigned to thread i

3.2 Composite Metric for Prefetching

Traditional metrics for prefetching performance include *coverage*, defined as the fraction of the original cache misses that are eliminated by prefetching; and *accuracy*, defined as the fraction of prefetch requests that successfully eliminate cache misses. These metrics cannot determine whether prefetching should be used or not in limited off-chip bandwidth environment because they do not take into account how much off-chip bandwidth is available, and how memory access latency is affected by prefetching.

To arrive at a new metric, we start from the basic *Cycle per Instruction* (CPI) model [6] and add the effect of additional queueing delay due to prefetching traffic. We use Δt to represent the queueing delay on the bus and let T_m be the average access latency to the memory [3], the CPI becomes:

$$CPI = CPI_{L2\infty} + h_m \cdot (T_m + \Delta t) \quad (2)$$

When prefetching is applied in the system, let CPI_p present the new CPI, Δt_p as the new queuing delay on the bus, $h_{m,p}$ as the new L2 misses per instruction, the CPI equation then becomes:

$$CPI_p = CPI_{L2\infty} + h_{m,p} \cdot (T_m + \Delta t_p) \quad (3)$$

Comparing Equation 3 vs. Equation 2, prefetching may decrease the L2 miss per instruction, i.e. $h_{m,p} < h_m$, but increase the queueing delay, i.e. $\Delta t_p > \Delta t$ due to the extra traffic generated by prefetch requests.

$h_{m,p}$ can be expressed as the multiplication of miss frequency (M_pA_p) and the average time taken to execute one instruction ($\frac{CPI_p}{f}$):

$$h_{m,p} = \frac{M_p \cdot A_p \cdot CPI_p}{f} \quad (4)$$

Substituting Equation 4 into Equation 3, and solving CPI_p, we can arrive at:

$$CPI_p = \frac{CPI_{L2\infty}}{1 - \frac{M_pA_p}{f}(T_m + \Delta t_p)} \quad (5)$$

Now we will derive Δt_p using Little's law. If we let λ_p denote the arrival rate of memory requests, then λ_p is equal to the sum of frequency of cache miss and prefetch requests:

$$\lambda_p = (M_p + P)A_p \quad (6)$$

[3]The average memory access latency T_m is an amortized value that implicitly includes the effect of memory bank and row buffer conflicts, Instruction Level Parallelism (ILP) as well as Thread Level Parallelism (TLP).

Little's law [12] for a queuing system states that the average queue length (L_p) is equal to the arrival rate (λ_p) multiplied by the service time (T_p), i.e. $L_p = \lambda_p \cdot T_p$, while the service time on the off-chip bus is the cache block size (K) divided by the available bandwidth (B). Hence, the average number of memory requests arriving during the service time is:

$$L_p = \frac{K}{B} \cdot (M_p + P)A_p \quad (7)$$

If we assume memory requests are not bursty, i.e. requests are processed back to back, the average waiting time of a newly arriving request is equal to the L_p requests that are ahead of it in the queue, multiplied by the service time. Thus, the waiting time Δt_p (in cycles) can be expressed as:

$$\Delta t_p = f \cdot \frac{K}{B} \cdot L_p = f \cdot \frac{K^2}{B^2} \cdot (M_p + P)A_p \quad (8)$$

Substituting Equation 8 into Equation 5, expression for CPI_p is:

$$CPI_p = \frac{CPI_{L2\infty}}{1 - \frac{M_pA_pT_m}{f} - \frac{M_p(M_p+P)A_p^2K^2}{B^2}} \quad (9)$$

Similarly, the system without prefetching has the CPI of:

$$CPI = \frac{CPI_{L2\infty}}{1 - \frac{MAT_m}{f} - \frac{M^2A^2K^2}{B^2}} \quad (10)$$

In order for prefetching to produce a net benefit in performance, the CPI after prefetching must be smaller than CPI without prefetching, i.e. $CPI_p < CPI$. From the definition of prefetching coverage and accuracy, we have (assuming L2 cache access frequency is not affected by prefetching, i.e. $A_p = A$):

$$c = \frac{MA - M_pA_p}{MA} = \frac{M - M_p}{M} \quad (11)$$

$$a = \frac{MA - M_pA_p}{PA_p} = \frac{M - M_p}{P} \quad (12)$$

Rearranging Equation 11 and Equation 12 to express M_p and P in terms of c and a, substituting them into the CPI_p expression in Equation 9, and simplifying the inequality $CPI_p < CPI$, we obtain the final expression for the inequality:

$$T_m > \frac{MAK^2}{B^2} f(c - 2 + \frac{1-c}{a}) \quad (13)$$

The right hand side of the inequality, $\frac{MAK^2}{B^2} f(c-2+\frac{1-c}{a})$, is the composite metric that takes into account prefetching coverage and accuracy, as well as cache block size, available bandwidth, and miss frequency. The left hand side of the inequality, T_m, measures the average exposed memory access latency. The inequality essentially provides a break-even point threshold for how large the average memory access latency should be for prefetching to be beneficial. It is easier to meet the inequality when the average memory access latency is large, the cache miss frequency is small, the available bandwidth is large, and the coverage and accuracy are large. All these factors make sense qualitatively, and Equation 13 captures their quantitative contributions. This leads us to:

OBSERVATION 1. *Whether prefetching improves or degrades performance cannot be measured just by its coverage or accuracy. Rather, a prefetching profitability criterion in Inequality 13 is needed, using input parameters that are easy to collect in hardware.*

Recall that in the model construction, we assumed memory requests are not bursty and requests are processed back to back. This allows us to reach bandwidth utilization of 100% if the arrival rate of memory requests is high. However, memory requests are bursty in reality, even high arrival rate cannot achieve 100% utilization. This introduces a small inaccuracy in our metric. Thus we can define a parameter $\alpha \in (0,1)$ such that:

$$T_m > \frac{MAK^2}{\alpha B^2} f(c - 2 + \frac{1-c}{a}) \tag{14}$$

α loosely represents the degree of burstiness of memory requests and can be obtained from empirical evaluation (as shown in Section 4.2.1). Since $\alpha < 1$, the minimum value of the right hand side of the inequality is $\frac{MAK^2}{B^2} f(c - 2 + \frac{1-c}{a})$. Thus we can conclude:

OBSERVATION 2. *If an application running on a core satisfies $T_m < \frac{MAK^2}{B^2} f(c - 2 + \frac{1-c}{a})$, prefetching is harmful to performance. In addition, if $0 > \frac{MAK^2}{B^2} f(c - 2 + \frac{1-c}{a})$, prefetching improves performance.*

3.3 Memory Bandwidth Partitioning Model

This model extends the bandwidth partitioning model from [7] by taking into account not just demand fetches, but also prefetch requests, and prefetching coverage and accuracy.

In Equation 5, we can compute the queueing delay Δt_p for thread i using Little's law for the case where we do not apply bandwidth partitioning (i.e. requests from all cores contend with each other for bandwidth access naturally) and compare it against the case where we apply bandwidth partitioning (i.e. bandwidth fraction is allocated to each core for its own requests). The CPI of thread i in system with natural sharing (no partitioning) and with bandwidth partitioning can be expressed as:

$$CPI_{p,i,nopt} = \frac{CPI_{L2\infty,i}}{1 - \frac{M_{p,i} A_i T_{m,i}}{f} - \frac{\left(\sum_{j=1}^{N} (M_{p,j}+P_j) A_j\right)^2 M_{p,i} K^2}{(M_{p,i}+P_i) B^2}} \tag{15}$$

$$CPI_{p,i,bwpt} = \frac{CPI_{L2\infty,i}}{1 - \frac{M_{p,i} A_i \cdot T_{m,i}}{f} - \frac{M_{p,i} (M_{p,i}+P_i) A_i^2 K^2}{\beta_i^2 \cdot B^2}} \tag{16}$$

where β_i denote the fraction of bandwidth allocated to thread i.

Comparing Equation 16 and 15, we can conclude that Equation 15 is a special case of Equation 16 where:

$$\beta_i = \frac{(M_{p,i} + P_i) A_i}{\sum_{j=1}^{N} (M_{p,j} + P_j) A_j} \tag{17}$$

which leads us to the following observation:

OBSERVATION 3. *In a CMP system where off-chip bandwidth usage among multiple cores is unregulated, the off-chip bandwidth is naturally partitioned between cores, where the natural share of bandwidth a core uses is equal to the ratio of memory request frequency (including both cache misses and prefetch requests) of the core to the sum of all memory request frequencies from all cores.*

Substituting CPI from Equation 16 into Equation 1, with constraint of $\sum_{i=0}^{N} \beta_i = 1$, we can maximize the weighted speedup using Lagrange multipliers [7]. The solution provides the bandwidth partition for thread i (β_i) is:

$$\beta_i = \frac{\left(C_i M_{p,i} (M_{p,i} + P_i) A_i^2\right)^{1/3}}{\sum_{j=1}^{N} \left(C_j M_{p,j} (M_{p,j} + P_j) A_j^2\right)^{1/3}} \tag{18}$$

where $C_i = \frac{CPI_{alone,i}}{CPI_{L2\infty,i}}$. This leads us to the next observation:

OBSERVATION 4. *Weighted speedup-optimum bandwidth partition for a thread can be expressed as a function of all co-scheduled threads' miss and prefetch frequencies, infinite-L2 CPIs, and CPIs when each thread runs alone in the system, as in Equation 18* [4].

[4]Parameters in Equation 18 may vary due to an application's phase change behavior. The equation does not assume them to be constant, but assumes the parameters for any specific execution interval are available.

4. EMPIRICAL EVALUATION

4.1 Environment and Methodology

Simulation Parameters. We use a cycle-accurate full system simulator based on Simics [15] to model a CMP system with dual and quad cores. Each core has a scalar in-order issue pipeline with 4GHz clock frequency. To remove the effect of cache contention between cores, each core has private L1 and L2 caches. The L1 instruction cache and data caches are 16KB, 2-way associative, and have a 2-cycle access latency. The L2 cache is 512KB, 8-way associative, and has an 8-cycle access latency. All caches use a 64-byte block size, implement write-back policy, and LRU replacement. The bus to off-chip memory is a split transaction bus, with a peak bandwidth of 800MB/s for a single core system, 1.6GB/s for dual-core CMP and 3.2GB/s for quad-core CMP. The average main memory access latency is 300 cycles. The CMP runs Linux OS that comes with Fedora Core 4 distribution.

Hardware Prefetchers. The L2 cache of each core is augmented with a typical stream prefetcher that is commonly implemented in modern processors [3, 8] due to their low hardware cost and high effectiveness for a wide range of applications. Stream buffers can detect accesses to block addresses that form a sequential or stride pattern (called a *stream*), and prefetch the next few blocks in the stream in anticipation of continuing accesses from the stream. We implement a stream prefetcher with four streams, and up to four blocks prefetched for each stream.

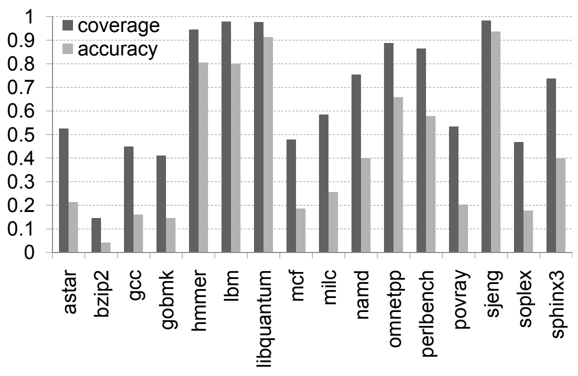


Figure 2: **Benchmarks prefetch coverage and accuracy.**

Workload Construction. We consider seventeen C/C++ benchmarks from the SPEC2006 benchmark suite [19]. We compile the benchmarks using `gcc` compiler with `O1` optimization level into `x86` binaries. We use the *ref* input sets, simulate 250 million instructions after skipping the initialization phase of each benchmark, by inserting breakpoints when major data structures have been initialized. We pair the benchmarks into co-schedules of two or four benchmarks. To reduce the number of co-schedules that we need to evaluate, we categorize the benchmarks based on the prefetching characteristics. Figure 2 shows the prefetch coverage and accuracy of each benchmark. From the figure, we pick up three representative benchmarks that have low (*bzip2*), medium (*astar*) and high (*hmmer*) coverage and accuracy, and pair each one with the other benchmarks to cover all representative co-schedules.

4.2 Experimental Results

4.2.1 Validating the Prefetching Metric

In this section, we investigate how well the composite prefetching metric in Equation 14 predicts prefetching performance profitability. We run each application on a single core system with available bandwidth of 800MB/s, and measure CPIs when prefetching is turned on (CPI_p) and when prefetching is turned off (CPI). Then we compute the difference ($\Delta CPI = CPI_p - CPI$) for each benchmark. A negative number represents performance improvement while a positive value represents degradation. Table 2 shows the applications sorted by their ΔCPIs. As a comparison, we compute our composite prefetching metric $\theta = \frac{MAK^2}{B^2} f(c - 2 + \frac{1-c}{a})$ and sort the applications in decreasing order of θ. Finally, conventional wisdom evaluates prefetching performance based on coverage or accuracy, so we also show the applications sorted in increasing order of accuracy. Note that Figure 2 shows that coverage and accuracy are highly correlated, hence we only show accuracy in the table. Applications whose ranks match with ones obtained from ΔCPI sorting are shown in bold.

Table 2: Ranking of applications based on ΔCPI, our composite metric θ, and prefetching accuracy a.

ΔCPI-sorted		θ-sorted		a-sorted
Benchmark	ΔCPI	Benchmark	θ	Benchmark
bzip2	4.837	**bzip2**	1375.77	**bzip2**
soplex	4.484	**soplex**	148.61	gobmk
mcf	4.463	**mcf**	145.88	gcc
gobmk	0.218	**gobmk**	85.44	soplex
gcc	0.207	**gcc**	78.25	mcf
povray	-0.006	astar	56.40	**povray**
namd	-0.012	milc	10.40	astar
omnetpp	-0.047	povray	0.83	milc
astar	-0.101	namd	-1.47	sphinx3
milc	-0.285	omnetpp	-7.92	namd
perlbench	-0.293	**perlbench**	-25.29	**perlbench**
hmmer	-1.364	**hmmer**	-43.11	omnetpp
sphinx3	-1.565	libquantum	-62.90	lbm
libquantum	-3.770	lbm	-69.39	hmmer
lbm	-4.196	sphinx3	-74.13	libquantum
sjeng	-13.820	**sjeng**	-76.86	**sjeng**

Let us compare the ranks of applications based on the actual performance improvement due to prefetching (ΔCPI), vs. based on our metric θ. The ranks of eight applications (shown in bold) out of sixteen exactly match. For the other eight applications, their ranks differ by at most three positions, indicating how well our metric θ correlates with the actual performance. If each application is given a rank number, and we compare ranks based on ΔCPI vs. θ, the correlation coefficient computes to 95%, indicating high correlation between them. In contrast, the correlation of ranks based on ΔCPI vs. accuracy a computes to 89%. Therefore, our composite metric correlates better with the actual performance than conventional metrics.

In addition, conventional metrics cannot determine what level of accuracy or coverage is high enough to produce performance improvement, whereas our composite metric θ includes a performance profitability threshold (Equation 14): $T_m > \frac{\theta}{\alpha}$, where $\alpha \in (0, 1)$ is a number that reflects how bursty memory accesses are. Even when α is unknown, θ can still determine prefetching profitability unambiguously in some cases. For example, $\theta = 1375.77$ for *bzip2*, which is much larger than the fully exposed memory access latency $T_m = 300$ cycles, hence θ predicts unambiguously that prefetching will hurt performance (Observation 2). In addition, there are eight applications showing negative θ values in Table 2. θ predicts unambiguously that prefetching will improve performance. Only for seven out of sixteen cases, the actual value of α is needed to determine whether prefetching is profitable.

If we choose $\alpha = 0.2$, all benchmarks ranked above *astar* satisfy $T_m \le \frac{\theta}{\alpha}$, while all benchmarks ranked at *astar* and below satisfy $T_m > \frac{\theta}{\alpha}$. Thus, 0.2 provides a reasonable estimate for α. For further validation, we run simulations on a single core with different prefetch algorithms and different bandwidth amount, all results point to the same conclusion that $\alpha = 0.2$ is a good estimate for SPEC2006 benchmarks.

4.2.2 Co-Deciding Prefetching and Partitioning

In this section, we show results where our prefetching profitability metric in Inequality 14 guides the decision to selectively turn on/off prefetchers, coupled with dynamic bandwidth partitioning.

The bandwidth partitioning is enforced using token bucket algorithm [7], where a token generator distributes tokens to different per-core buckets at the rates proportional to the fraction of bandwidth allocated to different cores. Each cache miss or prefetch request is allowed to go to the off-chip interface only when the core generating the request has matching tokens in its bucket. In the dynamic partitioning scheme, the bandwidth shares are re-evaluated every 1-million cycles. As in [7], we deploy a simplified practical implementation by assuming $C_i = 1$ in Equation 18.

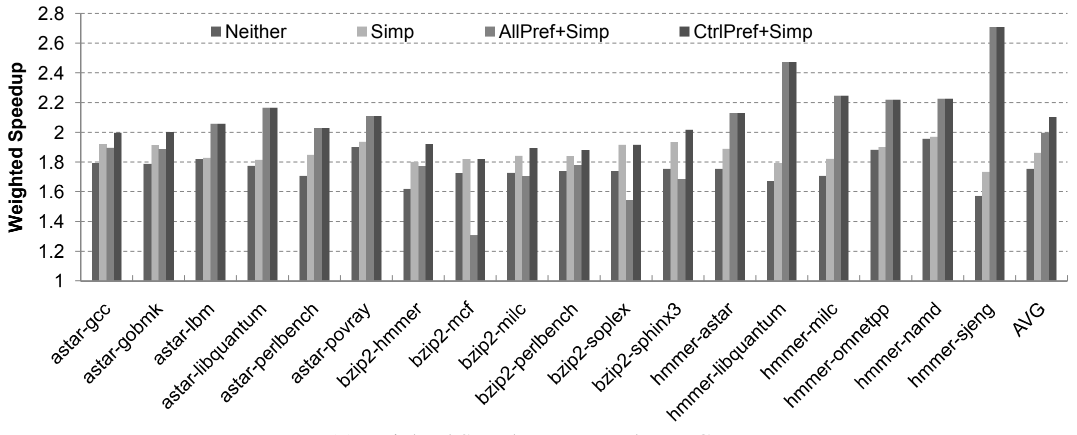
Figure 3 shows the weighted speedup for four system configurations a CMP system: no prefetching or bandwidth partitioning (Neither), only simplified bandwidth partitioning (Simp), all prefetchers turned on and bandwidth partitioning (AllPref+Simp), and selective activation of each core's prefetcher guided by prefetching profitability metric coupled with bandwidth partitioning (CtrlPref+Simp). Specifically, we assume $\alpha = 0.2$ as discussed in Section 4.2.1. To make prefetching decision, we first run the applications on a CMP system with equal share of the available bandwidth for each core, and profile the hardware counters needed for computing θ. Based on the computed θ, we statically turn on/off the prefetcher of a core, and run the applications again to collect performance. Note that nothing prevents the prefetching decision to be performed dynamically at run time. However, making prefetching decision statically is sufficient to demonstrate the effectiveness of the prefetching profitability metric. The experiments are evaluated on a dual-core CMP with available bandwidth of 1.6GB/s and on a quad-core CMP with available bandwidth of 3.2GB/s respectively.

The figure shows that in many cases, turning on all prefetchers (AllPref+Simp) degrades performance over no prefetching (Simp) (i.e. 8 out of 16 co-schedules on a dual core CMP, and in 3 out of 10 co-schedules on a quad-core CMP). Worse, performance of AllPref+Simp is even lower than the base case CMP without prefetching or bandwidth partitioning (Neither) in a majority of the above cases. This indicates that prefetching can be too harmful for bandwidth partitioning to offset.

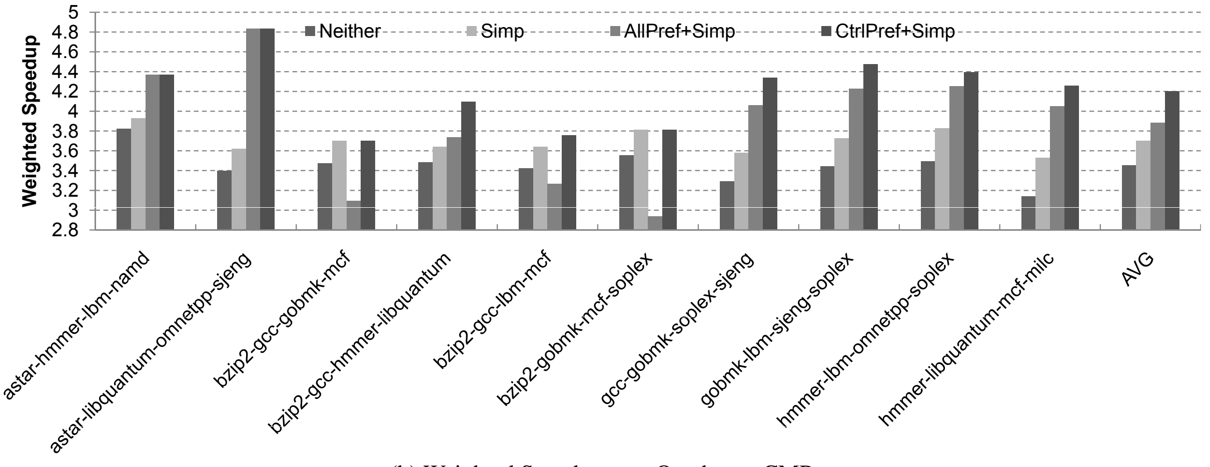
In both figures, when we use our prefetching profitability criterion to guide the decision to turn on/off each core's prefetcher, coupled with dynamic bandwidth partitioning, the best weighted speedup can be achieved for all co-schedules. CtrlPref+Simp has at least the same performance as the best of Simp and AllPref+Simp, and in many cases outperforms both of them significantly. On average, over Neither, Simp, and AllPref+Simp, CtrlPref+Simp improves weighted speedup by 6.0%, 13.6% and 19.5%, respectively on a dual-core CMP, and 7.2%, 12.5% and 21.8%, respectively on a quad-core CMP.

5. CONCLUSIONS

The goal of this paper is to understand how hardware prefetching and memory bandwidth partitioning in CMP can affect system performance and interact with each other. We have presented an analytical model to achieve this goal. Firstly, we derived a composite prefetching metric that can help determine under which situations hardware prefetcher can improve system performance. Then we constructed a bandwidth partitioning model that can derive weighted speedup-optimum bandwidth allocations among different cores. Finally, we collected simulation results to validate the observations obtained from the analytical model, and show system performance improvement that can be achieved by co-deciding prefetching and bandwidth partitioning decisions using our prefetching metric and implementable dynamic bandwidth partitioning.



(a) Weighted Speedup on a Dual-core CMP



(b) Weighted Speedup on a Quad-core CMP

Figure 3: Weighted speedup of optimum bandwidth partitioning for no prefetching, all prefetching, vs. selectively turning on prefetchers using the composite prefetching metric on dual-core CMP with 1.6GB/s bandwidth (a), and quad-core CMP with 3.2GB/s bandwidth (b).

6. REFERENCES

[1] A. Snavely et al. Symbiotic Job Scheduling for a Simultaneous Multithreading Processor. In *Proc. of 19th Intl. Conf. on Architecture Support for Programming Language and Operating Systems(ASPLOS)*, 2000.

[2] B. Rogers et al. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *Proc. of the 36th Intl. Conf. on Computer Architecture (ISCA)*, 2009.

[3] B. Sinharoy et. al. POWER5 System Microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.

[4] E. Ebrahimi et al. Coordinated Control of Multiple Prefetchers in Multi-Core Systems. In *Proc. of the 42th IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*, 2009.

[5] E. Ipek et al. Self-Optimizing Memory Controller: A Reinforcement Learning Approach. In *Proc.of the 35th Intl. Symp. on Computer Architecture (ISCA)*, 2008.

[6] P. Emma. Understanding Some Simple Processor-Performance Limits. *IBM Journal of Research and Development*, 41(3), 1997.

[7] F. Liu et al. Understanding How Off-Chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance. In *16th Intl. Symp. on High Performance Computer Architecture*, 2010.

[8] G. Hinton et al. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, (Q1), 2001.

[9] ITRS. International Technology Roadmap for Semiconductors: 2005 Edition, Assembly and packaging. In *http://www.itrs.net/Links/2005ITRS/AP2005.pdf*, 2005.

[10] K.J. Nesbit et al. Fair Queuing Memory System. In *Proc. of the 39th IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*, 2006.

[11] L. Spracklen et al. Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications. In *11th Intl. Symp. on High Performance Computer Architecture(HPCA)*, 2004.

[12] J. Little. A Proof of Queueing Formula L = λW. *Operations Research*, 9(383–387), 1961.

[13] N. Rafique et al. Effective Management of DRAM Bandwidth in Multicore Processors. In *Proc. of the 16th Intl. Conf. on Parallel Architectures and Compilation Techniques(PACT)*, 2007.

[14] O. Mutlu et al. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *Proc.of the 35th Intl. Symp. on Computer Architecture (ISCA)*, 2008.

[15] P.S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer Society*, 35(2):50–58, 2002.

[16] S. Rixner et al. Memory Access Scheduling. In *Proc.of the 27th Intl. Symp. on Computer Architecture (ISCA)*, 2000.

[17] S. Srikantaiah et al. SRP: Symbiotic Resource Partitioning of the Memory Hierarchy in CMPs. In *In Proc. of Intl. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC)*, 2010.

[18] S. Srinath et al. Feedback Directed Prefetching: Improving the Performance and Bandwidth-efficiency of Hardware Prefetchers. In *13th Intl. Symp. on High Performance Computer Architecture(HPCA)*, 2007.

[19] Standard Performance Evaluation Corporation. Spec cpu2006 benchmarks. *http://www.spec.org*, 2006.