Bandwidth Constrained Coordinated HW/SW Prefetching for Multicores

Sai Prashanth Muralidhara, Mahmut Kandemir, and Yuanrui Zhang

Department of Computer Science and Engineering
Pennsylvania State University, University Park, PA 16802, USA
{smuralid,kandemir,yuazhang}@cse.psu.edu

Abstract. Prefetching is a highly effective latency hiding technique that can greatly improve application performance. However, aggressive prefetching can potentially stress the off-chip bandwidth. The resulting bandwidth stalls can potentially negate the performance gain due to prefetching. In this paper, focusing on a multicore environment, we first study the comparative benefits of hardware and software prefetching and analyze if the two are complimentary or redundant. This analysis also evaluates different aggressiveness levels of hardware prefetching. Secondly, we weigh the positive performance benefits of prefetching against the negative performance effects of bandwidth stalls. Thirdly, we propose a hierarchical prefetch management scheme for multicores that controls the prefetch levels such that the overall performance gain is improved. Lastly, we show that our proposed off-chip bandwidth aware prefetch management scheme is very effective in practice, leading to performance gains of upto about 10% in system throughput over a bandwidth agnostic prefetching scheme.

1 Introduction

Prefetching is a well-known memory latency hiding technique, which predicts future memory accesses and proactively fetches the corresponding memory elements to the cache ahead of time in order to hide memory access latencies during execution [7] [8] [9] [10]. Prefetching can either be implemented at the hardware level [7] [8] [10] [9] or by the software [20] [18]. The effectiveness of a prefetching scheme is directly dependent on the predictability of memory accesses, which is an application characteristic. In a multicore system, each core prefetches data elements independently into the cache. The benefits due to prefetching can potentially be different for different cores depending on the application characteristics. Further, each core/application can potentially be involved in both hardware and software prefetching. There have been previous techniques proposed to throttle inaccurate prefetchers and increase aggressiveness levels on more accurate ones [28]. Also, when the last level cache is shared, aggressive prefetching can worsen the cache interference problem, especially when it is inaccurate and/or inefficient. In such cases, it is helpful to throttle the prefetches that are inaccurate and cause high interference in the shared cache space [11].

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 310–325, 2011.

In this paper, we first study the comparative accuracies and benefits of software prefetching and different levels of hardware prefetching. We then study and analyze the impact of prefetching on the off-chip memory bandwidth performance. Prefetching can lead to increased off-chip bus traffic, and can potentially increase the pressure on the off-chip bandwidth. This can cause extensive bandwidth stalls.We explore the tradeoff between extensive aggressive prefetching and bandwidth stalls. Further, we study if the performance degradation due to bandwidth stalls wipe away the performance gains achieved as a result of prefetching.

We propose a hierarchical bandwidth-aware coordinated prefetching scheme that manages the prefetch aggressiveness levels of different cores such that the performance gains due to prefetching are improved, while the performance losses due to bandwidth stalls are reduced. This prefetch management scheme operates dynamically and decisions are made at the end of each execution interval. More specifically, a *global prefetch manager* considers the overall bandwidth delay and the prefetch effectiveness of each core during each execution interval, and then decides to increase or decrease the prefetch aggressiveness levels of the cores. This decision to change the prefetch levels of the cores is made such that the performance improvement due to prefetching is higher than the stall time due to limited bandwidth and contention. It then directs the individual core-level prefetch managers to change the prefetch levels correspondingly. At each core, a *core-level prefetch manager* manages and enforces the prefetch aggressiveness levels. This prefetch manager not only issues hardware prefetch requests but also handles the software prefetch instructions. It decides whether to allow software prefetching or hardware prefetching or both and also at what aggressiveness levels. It is to be noted that prefetching on a core can be termed very aggressive if both hardware prefetching at the highest aggressiveness level and software prefetching is enabled. Aggressiveness can be downgraded by reducing the aggressiveness of hardware or software prefetching or both. Overall, the main goal of our approach is to reward useful prefetchers and punish the ones that hurt bandwidth availability without any performance benefit. Lastly, we evaluate our proposed scheme on set of workloads comprising of applications from the SPEC 2006 benchmark suite [1] on a simulation based setup, and show that our scheme yields average system throughput benefits of about 8%, and up to about 10% over an off-chip bandwidth unaware scheme. To summarize, we make the following contributions in this paper:

- We evaluate the performance benefit of both hardware (different levels) and software prefetching schemes. We later compare the performance improvement due to prefetching against the performance degradation due to the extra pressure it exerts on the off-chip bandwidth.
- We propose a *hierarchical prefetch management scheme* that tries to dynamically change the prefetch levels of the individual cores such that the performance degradation due to bandwidth contention is reduced and the performance improvement due to prefetching is improved.

• We present an extensive experimental evaluation of the proposed hierarchical prefetch management. Our results show that the proposed scheme is very effective in practice and improves the system throughput by up to 10%, and by an average of 8%.

2 Background and Methodology

2.1 Prefetching

Prefetching is a widely employed technique intended to improve on-chip cache performance [7] [8] [9] [10] [20] [18]. Prefetching, however, is not always beneficial. Some fraction of the predicted memory requests are never accessed. This is not the only instance of wasted prefetching. A future memory request prediction can turn out to be true but before the prefetched memory element is accessed, it might be evicted from the cache. Also, a prefetched request may kick out a useful data element from the cache. In these instances, prefetching increases the off-chip bus traffic and possibly cause bandwidth stalls without any significant benefit. Therefore, prefetch accuracy, which is an application characteristic determines the overall performance benefit from prefetching.

Hardware Prefetching. In the case of hardware prefetching, the future memory access prediction and the process of initiating requests to prefetch those elements are carried out by the hardware at runtime. Due to costs and limits on delay, hardware prefetchers generally implement a simple stride based prefetching or a stream based prefetching. A very aggressive hardware prefetcher would typically predict a large number of future memory requests and prefetch them. In comparison, a prefetcher with a lower aggressiveness level would be more conservative, predicting and issuing fewer prefetches. In this paper, we refer to and implement a stream prefetcher [28] [6] [24]. Aggressiveness level of a stream prefetcher is defined by two parameters: prefetch distance and prefetch degree [28] [6] [24]. *Prefetch Distance* dictates how far ahead of the demand access stream the prefetcher can issue prefetch requests, and *Prefetch Degree* determines how many cache blocks to prefetch when there is a cache block access to a monitored memory region.

Software Prefetching. In this case, the future memory access prediction is made statically, at compile time or at the coding time, and specific instructions are inserted into the code body to prefetch those predicted elements at the time of execution. Some applications render themselves to easy compile time prediction in which case the software prefetching is very effective [20] [18] [19]. Software prefetching also has the ability to employ complex and time consuming prefetching algorithms since the process is done apriori at compile time. Hardware prefetching, on the other hand, employs simpler prediction mechanisms but does well where software prefetching fails to analyze the code, e.g., as in the case of pointer-based applications.

2.2 Experimental Setup

Core architecture	UltraSparc 3, 3.1 GHz
Operating system	Sun Solaris 9
L1 caches	private, 3 cycle latency, direct-mapped
L2 cache	shared, 15 cycle latency, 16 way associative
Memory latency	260 cycles
Hardware Prefetcher	64 stream prefetcher per core, 4 prefetch levels
DRAM controller	demand-prefetch equal priorities, on-chip, 128 entry req buffer, FR-FCFS
DRAM chip	refer to Micron DDR2-800 [2]

Fig. 1. Default system parameters used

We model the off-chip memory bandwidth and implement the prefetching infrastructure for multicores using a Simics [3] based in-house module. The base system architecture simulated in our evaluations is a four-core multicore machine with a shared L2 cache and a shared off-chip memory bandwidth. The shared L2 cache is assumed to be a partitioned cache (i.e., its cache ways are distributed evenly across applications though in principle we could use any partitioning strategy). The cores simulated in this system are based on the UltraSparc 3 architecture [5]. The main architectural details of the simulated system are shown in the table given in Figure 1. In the evaluation of the proposed dynamic scheme later, we employ execution intervals of 10 million instructions. The hardware prefetcher used in this paper is a stream prefetcher [28] [6] [24] with 64 streams per prefetcher.

Benchmarks. For all the motivational and evaluation purposes, we use the applications from the SPEC 2006 benchmark suite [1], and construct our workloads from the subsets of these applications. To enable software prefetching on the applications, they are compiled on a SUN compiler with the highest optimization flag set.

Terminology. In this paper, by "prefetch level", we mean the "aggressiveness level" of the prefetcher. All types of prefetching mentioned in this paper are implemented for the last level of cache in a multicore. Whenever we refer to "software prefetching" in this paper, we mean the handling of the software-inserted prefetch instructions in the hardware. We do not propose or implement a new software prefetching algorithm. We compile the applications using a software prefetch enabled compiler that inserts prefetch instructions into the executable. We only refer to the way these instructions are handled in the hardware.

3 Empirical Motivation

3.1 Prefetching Benefits

The goal of this section is to compare the performance of various prefetching techniques with different aggressiveness levels across different applications.

Hardware Prefetching. Figure 2 plots the performance of our applications when different levels of prefetching are enabled compared to the case of no prefetching. We experimented with four different prefetch levels: *no prefetching, level 1, level 2* and *level 3*. *Level 1* prefetching has a prefetch distance of 4 and prefetch degree of 1. Prefetch distance and prefetch degree of *level 2* are

16 and 2 respectively, and those of *level 3* are 64 and 4. In this set of experiments, software prefetching is disabled, which means the prefetch instructions are ignored as no-ops. Since we are first interested in studying the performance benefits of prefetching in isolation, the performance effects due to bandwidth constraints are not considered in these experiments. From Figure 2, we can infer that while some applications are prefetch sensitive and, therefore benefit from more aggressive levels of prefetching, others do not exhibit large performance gains as prefetch levels are increased. In the above scenario, the prefetch levels can be reduced on applications that are not very prefetch-sensitive without a high performance penalty. On the flip side, increasing the prefetch levels on prefetch-sensitive applications can be very beneficial.

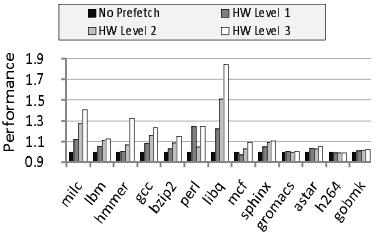

Fig. 2. Performance comparisons of different levels of hardware prefetching. The performance values are normalized to that of the no prefetching case.

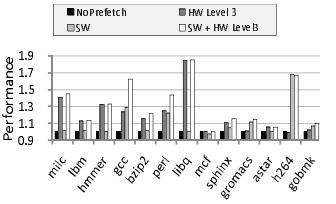

Fig. 3. Performance comparisons of software prefetching, hardware level 3 prefetching, and both with the case of no prefetching. The performance values are normalized to that of the no prefetching case.

Software Prefetching. Figure 3 compares software prefetching, hardware level 3 prefetching, combined software-hardware prefetching against the no-prefetching case. One can see from this plot that, for some applications, hardware prefetching does much better than software prefetching, whereas for some others, it is the other way around. More interestingly, in some cases, enabling both hardware and software prefetching is much better than enabling just one of them, as in the case of *gcc* and *perl*. In some other cases, although effective individually, enabling both does not do any better than enabling only one of them, as in the case of *astar* and *h264*. Therefore, in a multicore system, some applications perform better when both hardware and software prefetching are enabled, while some others perform equally well with just one of them enabled.

3.2 Off-Chip Bandwidth Effects

In this section, we study the effect of prefetching on off-chip bandwidth pressure. We employ an off-chip bandwidth of 6.4 GB/s in these experiments. For this purpose, we selected a workload of four applications: *lbm, mcf, libquantum,* and *milc*. These four applications are executed on a four-core processor (one application per core) with a shared, partitioned cache, and a shared off-chip bandwidth.

One core prefetching. In the first run, we enabled prefetching only on the first core which executed *lbm*, while disabling prefetching on all other cores. We

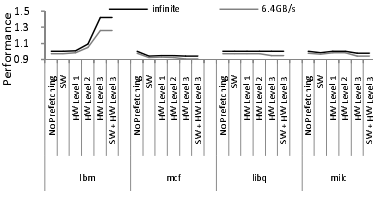

Fig. 4. Performance comparisons of different prefetching schemes with both the infinite bandwidth case and a bandwidth of 6.4 GB/s, when prefetching is enabled only on core 1 (*lbm*) and disabled for all others

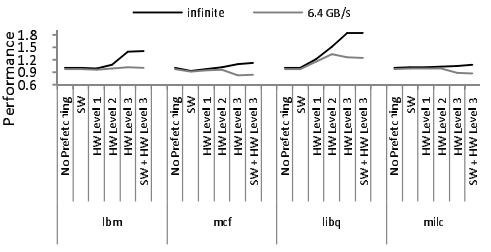

Fig. 5. Performance comparisons of different prefetching schemes with both the infinite bandwidth case and a bandwidth of 6.4 GB/s, when prefetching is enabled on all cores

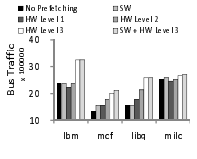

Fig. 6. Contributions to the bus traffic by different applications

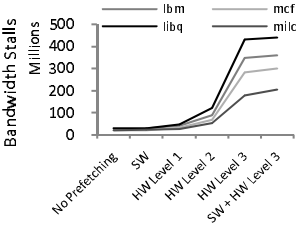

Fig. 7. Bandwidth stalls (in cycles) suffered by applications as the prefetching level is increased

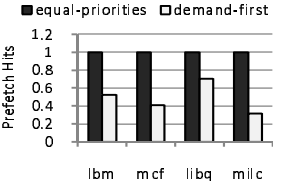

Fig. 8. Comparison of equal priorities for prefetch and demand requests versus a scheme where demand requests are prioritized over prefetch requests in terms of the number of useful prefetches

experimented with software prefetching, three levels of hardware prefetching, and a combined hardware-software prefetching scheme. The results in Figure 4 show that *lbm*, which executed on core 1, achieves a performance benefit when compared to the case of no prefetching. However, its benefits are reduced due to the limited bandwidth constraint. Further, it degrades the performance of the other applications due to the additional requests (prefetch requests from core 1) and the resulting bandwidth stalls. When using the most aggressive prefetching, the performance degradations on the other cores are significant. We also repeated this by enabling prefetching on core 2, core 3 and core 4 alone, and observed similar results. Therefore, prefetching can have different degrees of performance degradation due to bandwidth constraints. Further, *aggressive prefetching by one core can adversely impact the performance of other applications due to bandwidth contention and the resulting delays.*

All cores prefetching. We also considered a more realistic execution scenario, where different applications prefetch memory elements individually and the cores share the available off-chip bandwidth. In this case, prefetching is enabled on all

the cores. In Figure 6, we plot the comparative contributions to the bus traffic by the applications when prefetching is enabled on all the cores. The bus traffic increases rapidly when the prefetch level is increased for some applications, while for others, the increase is not that steep (for instance *milc*). Figure 7 shows how this increase in bus traffic translates into stalls due to limited bandwidth. Note that, even if the bus traffic increase is small, bandwidth stalls can be significant. The above two graphs plot absolute values of bus traffic increase and bandwidth stall cycles. Figure 5, on the other hand, illustrates how these factors affect the performance of applications when different prefetch variants are enabled for both the infinite bandwidth case and a more realistic case of 6.4 GB/s bandwidth. In the limited bandwidth case, prefetching aggressively in a bandwidth unaware manner on all the cores results in some performance improvement only on core 3 (*libq*). In all other applications/cores, performance degradation due to limited bandwidth completely wipes out all the benefits from prefetching and in some cases results in a net performance degradation. This effect increases with increasing prefetch levels. Also, for some applications, while absolute values of bandwidth stalls in Figure 7 increase sharply with prefetch levels, performance degradation is not that steep. Therefore, some applications are more bandwidth-stall resistant (tolerant). In modeling the performance effects later in Section 4.3, we take this into account. We do not just consider prefetch accuracies and the resulting bus traffic as the basis as done previously [28] [11] but also consider the bandwidth stalls and the actual impact of bandwidth stalls on application performance as the basis.

To summarize, while prefetching aggressively can improve performance, it can also hurt the performance due to bandwidth constraints. Therefore, it is important to enable prefetching without increasing bandwidth delays extensively.

3.3 Prefetch Request Priority

Increase in bandwidth delays due to prefetching typically occurs only if prefetch requests are treated on par with demand memory requests. If normal load/store (demand) memory requests have a higher priority than the prefetching requests, then additional bus traffic due to prefetch requests may not lead to any additional bandwidth delay. It is to be noted here that bandwidth delays might still be present in the system but those delays are due to the normal (demand) memory requests, and will be present irrespective of whether prefetching is turned on or not. Prioritizing demand requests and prefetching requests equally leads to increased performance improvement from prefetching as can be seen in Figure 8. This is due to the fact that if the prefetch requests have a lower priority than the demand requests, then the prefetch requests can get delayed inordinately and these increased bandwidth delays can render most of prefetch requests useless (since prefetched data would be brought into the cache late). This leads to decreased prefetch efficiency and, therefore, decreased positive performance impact of prefetching [16]. Therefore, our proposed scheme employs equal priorities, and tries to keep the number of useful prefetches high, while at the same time, mitigating the additional bandwidth stalls due to prefetch requests.

4 Bandwidth Aware Prefetching

Figure 9 summarizes the operation of our proposed scheme. A *global prefetch manager* makes decisions on whether to increase or decrease the prefetching levels on the individual cores and the decisions are communicated to the *core-level prefetch manager*.

The details on how these decisions are made are presented in Section 4.3. After the global manager directs a core-level prefetch manager to either increase or decrease the prefetch level of the core, the core-level manager applies the prefetch-level changes locally (i.e., to the core it is attached to), as described in Section 4.1. This prefetch management scheme works dynamically, making decisions on prefetch level changes and applying those changes at the end of each execution interval. This scheme is also history based, in the sense that all the relevant statistics, which include the total bandwidth stall-time and the prefetch efficiency counters of individual cores, collected during an execution interval are used to make decisions for the next execution interval.

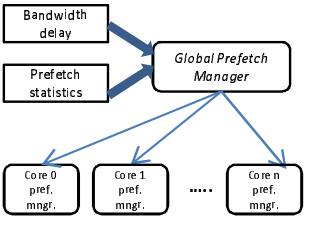

Fig. 9. Hierarchical bandwidth aware prefetching scheme that includes a *global prefetch manager* and a set of *core-level prefetch managers*

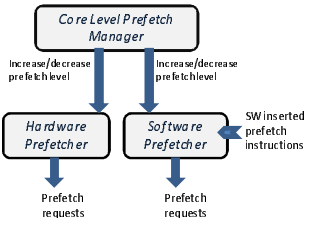

Fig. 10. Details of a core-level prefetch manager, which controls the prefetch levels of both hardware and software prefetchers of a core

Implementation. Hardware support is needed to maintain the performance counters. The prefetch management scheme itself is implemented in the runtime system/OS, which reads these hardware performance counters.

4.1 Core-Level Prefetch Manager

The core-level prefetch manager sets and enforces the prefetch aggressiveness level at the core level. It can either increase or decrease the prefetch level based on the directions from the global prefetch manager.

The core-level prefetch manager handles the changes in prefetch levels of the hardware prefetcher similar to that proposed in [11]. In addition to the hardware prefetcher, our proposed prefetch manager also employs a *software prefetcher*, which is an engine that handles all the software prefetch instructions issued by the core (compiler-inserted or programmer inserted). A prefetch instruction, when issued, results in a prefetch request. All such

```
increase_prefetch_level()
begin
  accuracy_HW = prefhits_SW / prefetches_HW
  accuracy_SW = Prefhits_SW / prefetches_SW
  if accuracy_HW > accuracy_SW
    //Increase HW prefetch level
    increase prefetch_distance_HW
    increase prefetch_degree_HW
  else
    //Increase SW prefetch level
    increase prefetch_distance_SW
    increase prefetch_degree_SW
end
```

Fig. 11. Prefetch level increase function

prefetch requests are routed through our proposed software prefetcher. When the global prefetch manager directs the core-level manager to either increase or decrease the prefetch level, the core-level manager can increase or decrease the prefetch level of either the hardware prefetcher or the software prefetcher. What we mean by "prefetch levels" in hardware and software prefetchers is explained later in detail. The role of the core-level prefetch manager in controlling the prefetch levels of both hardware and software prefetches is illustrated in Figure 10. The global prefetch manager either increases or decreases prefetch level, and does not set absolute values. The decision of whether to change the prefetch level of the hardware prefetcher or the software prefetcher is determined by calculating the corresponding *prefetch accuracies*. More accurate prefetcher is always preferred. This way, we prioritize either hardware or software prefetching based on their accuracies (the prefetch increase function is shown in Figure 11, prefetch decrease function is on similar lines).

4.2 Prefetch Levels

Hardware Prefetch Levels. We implement a stream prefetcher for hardware prefetching [6]. As mentioned earlier, the aggressiveness level of a stream prefetcher is defined by two parameters: *prefetch distance* and *prefetch degree*. Our hardware prefetcher design is similar to that implemented in [28] and further details on implementation can be found in [28] [6] [24]. In essence, the prefetch distance determines how far ahead of the memory stream the prefetch requests are issued and the prefetch degree determines how many prefetch requests are issued each time. We implement four prefetch levels in this work: *no prefetch*, *low prefetch*, *medium prefetch*, and *high prefetch*. No prefetch level performs no prefetching. Low prefetch level performs prefetching with a prefetch distance of 4 and prefetch degree of 1. Medium prefetch performs prefetching with a prefetch distance of 16 and a prefetch degree of 2, while the high prefetch level has prefetch distance of 64 and prefetch degree of 4.

Software Prefetch Levels. The software prefetcher implements the software prefetch levels by filtering the prefetch requests. As mentioned before, the software prefetcher receives all the prefetch requests that are issued by the software (compiler inserted or programmer inserted) instructions. The four aggressiveness levels of software prefetching are: *no prefetch*, *low prefetch*, *medium prefetch*, and *high prefetch*. When the level is set to no prefetch, all the prefetch requests are dropped. In the case of low prefetch level, two in every four prefetch requests are dropped, while only one in every four is dropped in the case of medium prefetch level. When the level is set to high prefetch, all prefetch requests coming from the software inserted prefetch instructions are issued by the software prefetcher without dropping any of them.

4.3 Global Prefetch Manager

As shown in Figure 9, the two main inputs to the *global prefetch manager* are the total bandwidth stall-time and the prefetch statistics.

Bandwidth stall-time. A demand request stalls in the memory controller queue if there are other requests ahead which are being serviced or waiting to be serviced. While the prefetch requests may also wait, they do not contribute to performance degradation (a higher wait-time for prefetch requests can of course limit the benefits due to prefetching). We define "*bandwidth_stall*" as the total stall-time (in cycles) experienced by the demand requests in a given execution interval. It is the sum of all individual demand request stall-times in that execution interval. Observe that "stall-time" in this context refers to wait-time in the queue due to bandwidth constraint. It does not include the time for a demand request to get serviced (to perform the memory operation). We compute *bandwidth_stall* using a simple counter in the memory controller. Since the off-chip bandwidth is a single resource shared across all the cores, *bandwidth_stall* is a single value, which is the sum of bandwidth stalls of all requests of all cores serviced by the off-chip bandwidth during the given execution interval.

Prefetch Statistics. As described in Section 4.1, each core has a hardware prefetcher and a software prefetcher associated with it. We define "$prefetches_i$" to be the total number of prefetches issued by core i. It is the sum of the number of prefetches issued by the hardware prefetcher and those issued by the software prefetcher. The metric "$prefhits_i$" is defined as the total number of prefetch requests (both hardware and software) that turned out to be hits for core i. These values are calculated using the prefetch bit of the cache line and by employing counters in the prefetchers.

Benefit Estimation. The performance improvement on core i due to prefetching is quantified by a parameter called "$benefit_i$". This improvement is specifically due to the avoidance of a fraction of core i cache misses. The metric $benefit_i$ is computed for each core i using the prefetch statistics collected during the execution interval as follows: $benefit_i = \frac{Reduction\_in\_cache\_miss\_stall\_time_i}{instructions_i}$. Therefore, accounting for this reduction in cache misses, we obtain: $benefit_i = \frac{(misses\_old_i - misses\_new_i) \times avg\_miss\_penalty}{instructions_i} = \frac{prefhits_i \times avg\_miss\_penalty}{instructions_i}$, where $instructions_i$ is the number of instructions executed in the current execution interval, $misses\_old_i$ is the estimated number of cache misses if prefetching was not enabled, $misses\_new_i$ is the number of cache misses with prefetching, and $avg\_miss\_penalty$ is the average cache miss penalty in cycles.

Cost Estimation. Prefetching leads to additional memory requests (in addition to the normal load/store demand requests). The measure of performance degradation suffered by core i due to memory bandwidth stall-time resulting from these prefetch requests it issues is quantified by the metric $cost_i$. Due to the fact that memory bandwidth is shared, the additional prefetches issued by core i can cause bandwidth stalls for not only core i but also for all other cores as well. As a result, $cost_i$ should take all these stalls into account. Firstly, the

total bandwidth stall caused by the prefetches issued by all the cores can be estimated as below: $total\_prefetch\_stall = \frac{\Sigma_{i=0}^{n} prefetches_i}{total\_requests} \times bandwidth\_stall$, where $\Sigma_{i=0}^{n} prefetches_i$ is the sum of prefetches issued by all the cores during the interval, $total\_requests$ is the total number of requests that reached the memory controller during the execution interval (i.e., sum of the demand and prefetch requests), and $bandwidth\_stall$ is the total bandwidth stall time as defined earlier. We can now estimate the stall caused by core i (due to the prefetches issued by core i) as follows: $prefetch\_stall_i = \frac{prefetches_i}{\Sigma_{i=0}^{n} prefetches_i} \times total\_prefetch\_stall$. For each core i, we now have $prefetch\_stall_i$, which is the estimated absolute bandwidth stall-time caused by the prefetch requests issued by core i. Since the off-chip bandwidth is a shared resource, $prefetch\_stall_i$, caused by core i can affect demand requests of any of the cores. We define $band\_stall_{i,j}$ as the bandwidth stall caused by the prefetches from core i on the performance of core j (on the demand requests of core j). This value estimates the fraction of the bandwidth stall of core j, due to the prefetch requests issued by core i. We can estimate $band\_stall_{i,j}$ as follows: $band\_stall_{i,j} = \frac{demand_j}{\Sigma_{i=0}^{n} demand_k} \times prefetch\_stall_i$, where $demand_j$ is the total number of demand requests issued by core j, which in this case is approximately equal to the number of L2 cache misses on core j, $\Sigma_{i=0}^{n} demand_k$ is the total number of demand requests issued by all cores.These $band\_stall_{i,j}$ values estimated above are the absolute stall times in cycles and not the impact on performance. Therefore, we now estimate $cost_i$, which is a measure of the total performance degradation caused by the prefetches issued by core i on the performance of all cores including core i. Note that performance degradation considered above is just the effect of bandwidth stalls. The value of $cost_i$ can be estimated as follows: $cost_i = \Sigma_{j=0}^{n} \frac{band\_stall_{i,j}}{instructions_j}$. It is important to note that, we do not consider prefetch accuracies or the absolute bandwidth stalls in our estimation of $benefit_i$ and $cost_i$ values. We estimate both these values in terms of the net effect on the application performance.

Algorithm. The global prefetch manager manages the prefetch levels for each core with the goal of improving the overall performance gains due to prefetching. In order to do so, global manager employs a cost/benefit analysis based scheme.

A prediction based dynamic scheme is employed by the global manager, i.e., the algorithm works by computing and making prefetch level changes for cores at the end of each execution interval. To begin with, all cores prefetch at the highest aggressiveness levels. The $benefit_i$ and $cost_i$ values are estimated for every core i at the end of each interval after

```
global_prefetch_manager()
begin
  for each execution interval:
    read bandwidth_stall
    for each i from 0 to num_cores:
      read instructions_i, prefetches_i and prefhits_i
      compute benefit_i and cost_i
      if (benefit_i − cost_i) >= cost_i × α then
        //increase the prefetch level of core i
        core_level_manager_i.increase_prefetch_level()
      else if (benefit_i − cost_i > 0 and
      benefit_i − cost_i < cost_i × α)
        //do not change the prefetch level of core i
      else (benefit_i − cost_i) <= 0 then
        //decrease the prefetch level of core i
        core_level_manager_i.decrease_prefetch_level()
  end for
end
```

Fig. 12. The algorithm executed by the global prefetch manager

reading the relevant performance counter values. For each core i, the prefetch level is increased if the $benefit_i - cost_i$ is greater than the $cost_i \times \alpha$ (i.e., if $benefit_i$ is greater than $cost_i$ by α percentage). If, on the other hand, the $benefit_i - cost_i$ is lower than the $cost_i \times \alpha$ but greater than zero, then the prefetch level is left unchanged. Finally, if $benefit_i$ is less than the $cost_i$ value, then the prefetch level is decreased for core i. The global prefetch manager enforces the prefetch level change for a given core i by directing the core-level manager of the corresponding core. The reason for reducing the prefetch level for a given core is obvious since the estimated benefit is lower than the estimated cost. On the other hand, increasing the prefetch level is more nuanced. The level is increased only if the estimated benefit is greater than the cost by a *pre-defined threshold value* (α).If the benefit is not greater than the cost by α percentage, the prefetch level is left unchanged. This algorithm can reduce the prefetch level of a core i gradually to zero (which means no prefetches are issued) when $benefit_i$ continues to be lesser than $cost_i$ after continuous prefetch level decrements. In this case, when the prefetch level is zero, $benefit_i$ will always be zero and the prefetch level will potentially be stuck at zero without being increased. To avoid this scenario, the core-level prefetch manager increments the prefetch level of a core to level 1 if the prefetch level is stuck at zero for more than two execution intervals. In this algorithm, since we consider benefit and cost values in terms of estimated changes in application performance, the goal is always to improve the performance of applications and improve the overall system throughput.

α values. The α values are tunable to make the prefetching scheme more conservative or more aggressive. We experimented with a lot of α values and finally determined that a value of 0.2 is reasonable. Therefore, in our implementation, if the benefit exceeds the cost by 20%, we increase the prefetch level.

5 Experimental Evaluation

Our evaluation setup is described in Section 2. A four-core machine with a shared, partitioned L2 cache was modeled as the underlying multicore architecture. We built several workloads that consist of four applications, each from the SPEC 2006 suite [1]. In all our evaluations, we collect results and data for a period of 1 billion cycles. Cache is however warmed up for a period of 500 million instructions prior to collecting results. We consider execution intervals of 10 million instructions. Our proposed prefetching scheme is called *Dyn_Band* throughout the experimental section.

Average Throughput. Figure 13 presents the throughput gain acheived by our proposed scheme (*Dyn_Band*) over other prior prefetching schemes when averaged over 10 different workloads we experimented with. Different workloads might benefit differently from the prior prefetching schemes. Our proposed scheme recognizes this and enables only those prefetching schemes and levels that benefits the workloads, also taking into account the bandwidth pressure exerted by the extra prefetch memory requests. Our proposed scheme yeilds an average

system throughput gain of about 8% over the best of the previous prefetching schemes.

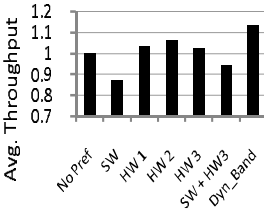

Fig. 13. Comparison of workload throughput averaged across multiple workloads

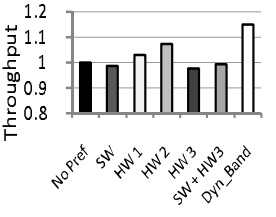

Fig. 14. Throughput comparison for the workload (*lbm*, *mcf*, *libquantum*, and *milc*).

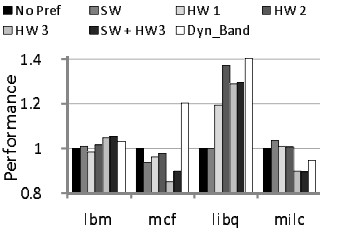

Fig. 15. Performance comparisons of the applications in the workload (*lbm*, *mcf*, *libquantum*, and *milc*).

Workload Instance. In order to understand our proposed scheme in more detail, we now present the results for a single workload instance that consists of *lbm*, *mcf*, *libquantum*, and *milc*. The corresponding throughput results are shown in Figure 14. In this case, our proposed dynamic bandwidth-aware prefetching scheme improves throughput by 15% over the no prefetching scheme. Among the other prefetching schemes, hardware level 2 prefetching does better than others because of lower pressure on off-chip bandwidth. Our dynamic bandwidth-aware scheme has a throughput gain of about 8% over this hardware level 2 prefetching. Figure 15 shows the individual application performance values. We observe that the application *milc* gains about 40% in performance over the no prefetching scheme and *mcf* gains about 20%.

Dynamics of the system. In order to analyze the working of our proposed scheme, we consider the execution of a workload comprising of *bzip2, libq, sphinx* and *gromacs* applications, and focus on the performances of *libq* and *gromacs*. We track how our scheme works dynamically, and adjusts the prefetch levels of these two applications based on their $benefit$ and $cost$ values (note here that our scheme works and adjusts the prefetch levels of all four applications; we focus on just two for clarity).

Figures 16 and 17 plot the observed $benefit$ and $cost$ values for these two applications for 11 execution intervals, when our scheme is used. In the case of *libq*, the $benefit$ value is consistently higher than the $cost$ value, while in the case of *gromacs*, the values are very close together. In order to study how our scheme dynamically changes the prefetch levels in accordance with the above values, we plot the $\frac{benefit-cost}{cost}$ values for the two applications for the same 11 execution intervals in Figure 18. Recall that, in the global prefetch management algorithm presented earlier in Figure 12, the equation $benefit_i - cost_i > cost_i \times \alpha$ is used to decide whether to increase the prefetch level or not. If the value $\frac{benefit-cost}{cost}$ is greater than α (0.2), then the prefetch level is increased and so on. Figure 19 plots the prefetch level changes made by our proposed scheme for both the applications. Note that, at execution interval 3, the prefetch level of *gromacs*

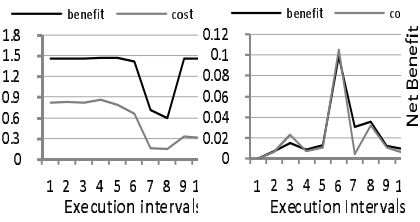

Fig. 16. Benefit and cost values of *libq* during execution

Fig. 17. Benefit and cost values of *gromacs* during execution

Fig. 18. Net benefit values of *libquantum* and *gromacs* during execution

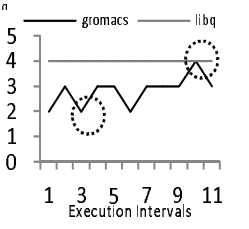

Fig. 19. Prefetch levels of *libquantum* and *gromacs* during execution

is reduced to 2 because the $benefit - cost$ value is less than zero (circled in Figures 18 and 19). Also, at execution interval 10, when $\frac{benefit-cost}{cost}$ becomes greater than 0.2 for gromacs, the prefetch level is increased to 4. However, it is reverted back because it was not highly benefitial. On the flipside, the prefetch level of *libq* is maintained at 4 since its $\frac{benefit-cost}{cost}$ values are consistently greater than 0.2.

Sensitivity analysis. We increased the memory bandwidth from 6.4 GB/s to 12.8 GB/s and executed the workloads. An average throughput improvement of about 7% over the best other prefetching scheme was observed. Therefore, even with higher bandwidth, our scheme achieves significant throughput improvement. We also experimented with different α values and found that a value of 0.2 provides the right balance.

6 Related Work

Hardware Prefetching. Hardware-controlled prefetching is an efficient way to implement prefetching [15] [7] [8] that tries to mitigate the negative effect of cold misses. Sequential prefetching automatically prefetches several consecutive data blocks into the cache upon a miss in the cache [9] [10]. Palacharla and Kessler investigate advanced stream buffers and filtering techniques to enhance the prefetching efficiency [24]. Hur and Lin discuss a dynamic stream detection technique that adapts the aggressiveness levels of prefetching in order to improve prefetching performance [13].

Software Prefetching. Seminal work related to software prefetching was authored by Mowry et al in [20], where they propose to use software controlled prefetch instruction insertion to enable prefetching. Other software prefetching schemes include [18] [20].

Prefetch Control. Srinath et al propose to use feedback control to improve the positive impact of prefetching and mitigate the adverse impact of harmful prefetches [28]. In [11], Ebrahimi et al investigate a control mechanism that can dynamically adjust the prefetch aggressiveness levels.

Off-Chip Bandwidth Studies. Rixner et al [27] introduce a scheduling policy that favors requests that hit in the row buffer over other requests. Nesbit et al suggest to prioritize memory requests of applications in accordance to their QoS requirements [23]. Rafique et al propose to adaptively change the fraction of memory bandwidth allocation for each thread [25]. In [14], Ipek et al study a machine learning approach in which a reinforcement learning based scheme is used to dynamically adapt scheduling decisions in the memory controller. Mutlu and Moscibroda proposed a stall time fair memory access scheduling in [21] and a parallelism-aware batch scheduling scheme in [22]. Liu et al study the effects of memory bandwidth partitioning on system performance [17].

Prefetching and Off-Chip Bandwidth. Lee et al propose to dynamically increase and decrease the priorities of prefetch requests at the memory controller in order to improve the benefits due to prefetching and decrease the penalties of inaccurate prefetchers [16]. In [12], Ebrahimi et al introduce a cooperative hardware/sofwtare approach to prefetch linked date structures in a bandwidth-efficient way.

In this paper, we considered the off-chip bandwidth as an important constraint, based on which, the prefetching levels of different cores are adjusted such that the prefetch benefits are improved. We considered the off-chip bandwidth stalls instead of the inter-core interferences [11] as the constraint. We did so because inter-core interferences are not prefetch specific and can result from demand accesses as well. We also modeled the benefits and costs of prefetching in terms of performance changes in this work, which makes our scheme throughput driven, and evaluated the comparative benefits of hardware and software prefetching.

7 Concluding Remarks

In this paper, we proposed a smart prefetch management scheme that exploits the performance benefits of prefetching while mitigating the performance degradation due to bandwidth stalls. Our proposed scheme is very effective in practice yielding a performance benefit of up to 8% in throughput over a bandwidth unaware prefetching strategy.

References

1. http://www.spec.org/spec2006
2. Micron: 1GB DDR2 SDRAM component: MT47H128M8HQ-25, http://download.micron.com/pdf/datasheets/dram/ddr2/1GbDDr2.pdf
3. Magnusson, P.S., et al.: Simics: A full system simulation platform. Computer 35(2), 50–58 (2002)
4. Xie, Y., Loh, G.H.: Dynamic Classification of Program Memory Behaviors in CMPs. In: CMP-MSI (2008)
5. Hetherington, R.: The UltraSparc T1 processor. SUN (2005)

6. Tendler, J., et al.: Power4 System Microarchitecture. IBM Technical White Paper (October 2001)
7. Baer, J.-L., Chen, T.-F.: An effective on-chip preloading scheme to reduce data access penalty. In: Proc. SC (1991)
8. Charney, M.J., Puzak, T.R.: Profetching and memory system behavior of the spec95 benchmark suite. IBM J. Res. Dev. (1997)
9. Dahlgren, F., et al.: Fixed and adaptive sequential prefetching in shared memory multiprocessors. In: Proc. ICPP (1993)
10. Dahlgren, F., et al.: Sequential hardware prefetching in shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. (1995)
11. Ebrahimi, E., et al.: Coordinated control of multiple prefetchers in multi-core systems. In: Proc. MICRO (2009)
12. Ebrahimi, E., et al.: Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In: Proc. HPCA (2009)
13. Hur, I., Lin, C.: Memory prefetching using adaptive stream detection. In: Proc. MICRO (2006)
14. Ipek, E., et al.: Self-optimizing memory controllers: A reinforcement learning approach. In: Proc. ISCA (2008)
15. Jouppi, N.P.: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. SIGARCH Comput. Archit. News (1990)
16. Lee, C.J., et al.: Prefetch-aware dram controllers. In: Proc. MICRO (2008)
17. Liu, F., et al.: Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In: Proc. HPCA (2010)
18. Mowry, T., Gupta, A.: Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. J. Parallel Distrib. Comput. (1991)
19. Vanderwiel, S., Lilja, D.: Data Prefetch Mechanisms. ACM Computing Surveys, CSUR (2000)
20. Mowry, T.C., et al.: Design and evaluation of a compiler algorithm for prefetching. In: Proc. ASPLOS (1992)
21. Mutlu, O., Moscibroda, T.: Stall-time fair memory access scheduling for chip multiprocessors. In: Proc. MICRO (2007)
22. Mutlu, O., Moscibroda, T.: Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In: Proc. ISCA (2008)
23. Nesbit, K.J., et al.: Fair queuing memory systems. In: Proc. MICRO (2006)
24. Palacharla, S., Kessler, R.E.: Evaluating stream buffers as a secondary cache replacement. In: Proc. ISCA (1994)
25. Ebrahimi, E., et al.: Fairness via source throttling: a configurable and high-performance fairness substrate for multicore memory systems. In: Proc. ASPLOS (2010)
26. Rafique, N., et al.: Effective management of dram bandwidth in multicore processors. In: Proc. PACT (2007)
27. Rixner, S., et al.: Memory access scheduling. In: Proc. ISCA (2000)
28. Srinath, S., et al.: Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In: Proc. HPCA (2007)