A Compiler-Directed Data Prefetching Scheme for Chip Multiprocessors [*]

Seung Woo Son
Pennsylvania State University
sson@cse.psu.edu

Mahmut Kandemir
Pennsylvania State University
kandemir@cse.psu.edu

Mustafa Karakoy
Imperial College
m.karakoy@yahoo.co.uk

Dhruva Chakrabarti
HP Labs
dhruva.chakrabarti@hp.com

Abstract

Data prefetching has been widely used in the past as a technique for hiding memory access latencies. However, data prefetching in multi-threaded applications running on chip multiprocessors (CMPs) can be problematic when multiple cores compete for a shared on-chip cache (L2 or L3). In this paper, we (i) quantify the impact of conventional data prefetching on shared caches in CMPs. The experimental data collected using multi-threaded applications indicates that, while data prefetching improves performance in small number of cores, its benefits reduce significantly as the number of cores is increased, that is, it is not scalable; (ii) identify harmful prefetches as one of the main contributors for degraded performance with a large number of cores; and (iii) propose and evaluate a compiler-directed data prefetching scheme for shared on-chip cache based CMPs. The proposed scheme first identifies program phases using static compiler analysis, and then divides the threads into groups within each phase and assigns a customized prefetcher thread (helper thread) to each group of threads. This helps to reduce the total number of prefetches issued, prefetch overheads, and negative interactions on the shared cache space due to data prefetches, and more importantly, makes compiler-directed prefetching a scalable optimization for CMPs. Our experiments with the applications from the SPEC OMP benchmark suite indicate that the proposed scheme improves overall parallel execution latency by 18.3% over the no-prefetch case and 6.4% over the conventional data prefetching scheme (where each core prefetches its data independently), on average, when 12 cores are used. The corresponding average performance improvements with 24 cores are 16.4% (over the no-prefetch case) and 11.7% (over the conventional prefetching case). We also demonstrate that the proposed scheme is robust under a wide range of values of our major simulation parameters, and the improvements it achieves come very close to those that can be achieved using an optimal scheme.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers, Optimization; C.1.4 [*Processor Architecture*]: Parallel Architectures

General Terms Algorithms, Experimentation, Performance

Keywords Chip multiprocessors, compiler, prefetching, helper thread

[*] This work is supported in part by NSF grants CCF #0702519, CNS #0720749, CNS #0720645, CCF #0811687 and a grant from Microsoft Research.

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA.
Copyright © 2009 ACM 978-1-60558-397-6/09/02...$5.00

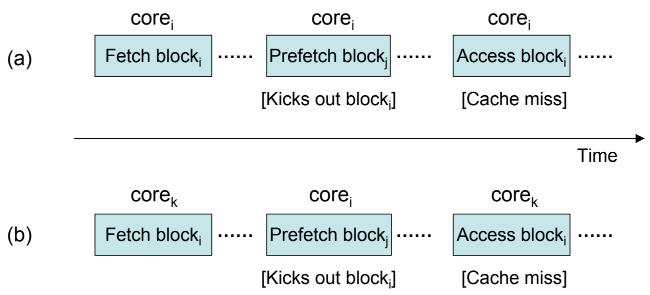


Figure 1. Example of intra-core (a) and inter-core (b) harmful prefetches.

1. Introduction

Prefetching has been shown to be a very effective technique for improving performance by hiding memory access latencies. However, timing and scheduling of prefetch instructions is a critical issue in software data prefetching and prefetch instructions must be issued in a timely manner for them to be useful. If a prefetch is issued too early, there is a chance that the prefetched data will be replaced from the cache before its use or it may also lead to replacement of other useful data from the higher levels of the memory hierarchy. If the prefetch is issued too late, the requested data may not arrive before the actual memory reference is made, thereby introducing processor stall cycles.

Software data prefetching (implemented through compiler-inserted explicit prefetch instructions) is inherently less speculative in nature than its hardware counterpart. However, scheduling prefetch instructions is the key for the success of any software prefetching algorithm. The criticality of scheduling the prefetch instructions increases in emerging chip multiprocessors (CMPs) where multiple cores prefetch data to the shared on-chip L2/L3 caches, due to the additional possibility of negative interactions among different processor cores.

The latest versions of many architectures employ some form of chip multiprocessing with a shared L2/L3 cache (20; 18; 17; 26; 23; 35). In these CMPs, the cores compete for the cache as any other shared resource. In the context of CMPs with shared on-chip caches, existing compiler algorithms for scheduling software prefetch instructions and existing techniques to compute prefetch distances may not be very effective. This is because prefetches from different cores to the same shared on-chip cache can pollute this cache, i.e., data brought to the cache by one core can be kicked out from the cache by another core. As a result, uncoordinated prefetch requests from different cores can significantly reduce cache utilization and lead to performance loss. This is unfortunate as the shared L2/L3 cache is the last line of defense before off–chip memory accesses in these systems and therefore achieving a high accuracy/success for data prefetches is of critical importance.

We call a prefetch *harmful* if the prefetched data is used later than the data it displaces from the on-chip cache. Note that, harmful prefetches can occur among the accesses made by the same core, or among the accesses from different cores, as illustrated in Figure 1. Harmful prefetches that involve a single core are referred to as intra-core (as shown in Figure 1(a)) harmful prefetches whereas those involving different cores are called inter-core (as shown in Figure 1(b)) harmful prefetches. Reducing the number of harmful prefetches can lead to a better utilization of the shared cache space and improve overall performance. Motivated by this observation, this paper makes the following contributions:

- We quantify the impact of harmful prefetches in the context of shared L2 based CMPs, when existing compiler-directed prefetching is used. Our experiments with several data-intensive multi-threaded applications indicate that the effectiveness of the conventional compiler-directed prefetching (which allows each core to perform its own prefetches independently) drops significantly as we move from single-core execution to multi-core execution. For example, in two of our applications (gafort and ammp), the performance with conventional data prefetching is even worse than the performance with the no-prefetch case beyond a certain number of cores.
- We show that the contribution of harmful prefetches also increases with the increased number of cores. And, therefore, there is a correlation between the degradation in the effectiveness of prefetching and the increase in the fraction of harmful prefetches, when the number of cores is increased. For instance, in applications like galgel, swim, and mgrid, the fraction of harmful prefetches can be as high as 20% beyond a core count.
- We demonstrate that data access and sharing patterns of a given multi-threaded application can be divided into distinct phases, and each phase typically requires a different prefetching strategy than the others for the maximum performance.
- We propose a novel, compiler directed data prefetching scheme that targets shared cache based CMPs. In this scheme, referred to as *L2 aware prefetching* or *helper thread based prefetching*, the compiler identifies program phases using static analysis and develops a customized data prefetching scheme for each phase. This scheme partitions, for each phase, threads into groups, and assigns a *helper thread* to each group. The assigned helper thread performs prefetching on behalf of all the threads in its group, and the application threads themselves do not issue prefetch requests.
- We present experimental evidence demonstrating the effectiveness of our approach. The results obtained using full system simulation indicate that our approach reduces the number of harmful prefetches significantly. As a result, the proposed scheme improves overall parallel execution latency by 18.3% over the no-prefetch case and 6.4% over the conventional data prefetching scheme (where each core prefetches its data independently), on average, when 12 cores are used. The corresponding average performance improvements with 24 cores are 16.4% (over the no-prefetch case) and 11.7% (over the conventional prefetching case). We also show that the proposed scheme is robust under a wide range of values of our major simulation parameters, and the improvements it achieves come very close to those that can be achieved using an optimal scheme.

The next section briefly explains the baseline data prefetching scheme, and presents an experimental evaluation of it in the context of CMPs using multi-threaded applications. Our new data prefetching scheme is explained in Section 3, and evaluated experimentally in Section 4. Related work is discussed in Section 5, and the paper is concluded in Section 6.

2. An Evaluation of the Conventional Prefetching for CMPs

In this section, we first explain the conventional compiler-directed data prefetching scheme against which our new scheme is compared and then evaluate its performance in the context of CMP. The baseline software based data prefetching approach used in this paper is similar to that proposed in (37). In this approach, which is developed in the context of single core systems, prefetches are inserted into the code based on data reuse analysis. More specifically, an optimizing compiler analyzes the application code and identifies future accesses to data elements that are not likely to be in the data cache. It then inserts explicit prefetch instructions to bring such elements into the data cache ahead of time to ensure that data is in the cache when it is actually referenced. As a result, a successful data prefetch effectively hides the off-chip memory latency.

```
int a[N];
int b[N];
int c[N];
for (i = 0; i < N; i++) {
  c[i] = a[i] * b[i];
}
```

(a)

```
int a[N], b[N], c[N];
prefetch(&a[0], D);    prefetch(&b[0], D);
for (t = 0; t < ⌊N/D⌋; t++) {
  prefetch(&a[(t + 1) * D], D);
  prefetch(&b[(t + 1) * D], D);
  for (i = 0; i < D; i++)
    c[t * D + i] = a[t * D + i] * b[t * D + i];
}
for (j = ⌊N/D⌋ × D; j < N; j++)
  c[j] = a[j] * b[j];
```

(b)

Figure 2. An example that illustrates compiler-directed data prefetching. (a) Original code fragment. (b) Code with explicit prefetch instructions targeting arrays a and b inserted. D represents the unit for prefetching.

Figure 2 illustrates an example application of this prefetching scheme. In this example, three N-element arrays (a, b, and c) are accessed using three references ($a[i]$, $b[i]$, and $c[i]$). D denotes the block size, which is assumed to be the unit for data prefetching (i.e., a prefetch targets a data block of size D). Figure 2(a) shows the original loop (without any data prefetching), and Figure 2(b) illustrates the compiler-generated code with explicit prefetch instructions embedded. The original loop is modified to operate on a block size granularity. As can be seen in the compiler generated code of Figure 2(b), the outermost loop iterates over individual blocks, whereas the innermost loop iterates over the elements within a block. This way, it is possible to prefetch a data block and operate on the elements it contains. The first two prefetch statements in the optimized code are used to load the first data blocks into the cache prior to loop execution. In the steady state, we first issue prefetch requests for the next set of blocks and then operate on the current set of blocks. The last loop nest is executed separately as the total number of remaining data elements may be smaller than a full block size.

This compiler based data prefetching scheme has two components: compiler component and runtime system support. The compiler component analyzes the given application code and predicts the future data access patterns. This is done using data reuse analysis, a technique developed originally in the context of cache locality optimization (54). After that, potential cache misses are isolated through loop-splitting and prefetches are pipelined based on the data reuse information generated by the compiler. In deciding the loop splitting point, the prefetching algorithm takes into account the estimated off-chip memory latencies as well (i.e., the time it takes to bring the data from the off-chip memory to the on-chip cache). The runtime layer on the other hand monitors the prefetch requests made by the application, and filters unnecessary prefetches as much as possible. For this purpose, the runtime layer maintains a fixed-size FIFO list that holds the most recent prefetch addresses. Therefore, whenever there is a new prefetch request to the L2 cache, it compares the requested address with the ones in the list. If there is a matching entry, that request is discarded; otherwise, it is allowed to proceed. By doing so, we attempt to minimize the cost of useless/duplicate prefetch requests. Note that, in a CMP architecture, this prefetch filtering component can be implemented in two different ways: per core or per chip. When the filtering module is implemented per core, each core has its own filtering module and performs individual prefetch filtering, independent of the other cores. When the filtering module is implemented per chip on the other hand, there is only one filtering module that does prefetch filtering. Clearly, the second option can filter

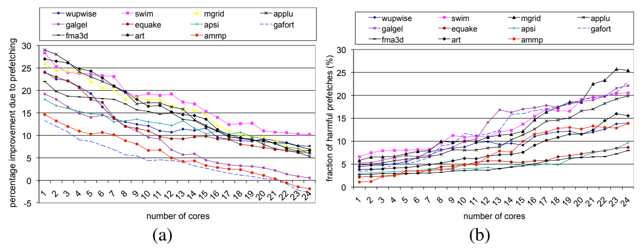


Figure 3. Behavior of the compiler-directed prefetching scheme in (37) when used in a CMP. (a) Performance improvement due to data prefetching under different core counts (savings are over the no-prefetch case). (b) Fraction of harmful prefetches under different core counts.

more duplicate prefetches. In this case however, the FIFO list becomes a shared entity and updates to it should be controlled using a semaphore or a similar mechanism.

In order to see how this conventional compiler-directed prefetching (also called *independent prefetching* in this paper) performs when applied to a multi-threaded application running on a CMP, we performed a set of experiments. Note that we compare our approach to this alternate prefetching scheme in which each core performs its own prefetching. In this initial set of experiments, the per core prefetch filtering has been used (later we also present results with the per chip prefetch filtering). The results from these experiments are presented in Figure 3.[1] In Figure 3(a), we quantify the percentage improvements this conventional data prefetching scheme brings over the no-prefetch case under varying core counts (from 1 to 24). Our main observation is that the improvements brought by this independent prefetching scheme keep reducing as we increase the number of cores. In fact, it is interesting to note that, in two applications (gafort and ammp), the prefetched version performs even worse than the no-prefetching case beyond 21 cores. Clearly, independent prefetching does not scale well for CMPs. We also want to mention that this independent prefetching scheme represents current state-of-the-art as far as single core based prefetching schemes are concerned. In fact, the performance of this independent prefetching scheme was very similar to the commercial prefetching scheme used in SUN's compiler (48) (the difference between two schemes was less than 3% for all the applications tested).

The results in Figure 3(b) provide an answer for why this trend occurs. This graph plots the fraction of data prefetches that are harmful. These results represent both intra- and inter-core harmful prefetches and nearly 72% of these harmful prefetches are inter-core ones. Recall that we refer to a prefetch as *harmful* if the prefetched data displaces a data block from the cache whose next usage is earlier than that of the prefetched block. We see that the contribution of harmful prefetches increases as we increase the number of cores. In fact, for some applications such as galgel, swim and mgrid, the fraction of harmful prefetches is above 20% beyond a certain core count. Now, putting the results in Figures 3(a) and (b) together, we can see the correlation[2] between harmful prefetches and degradation in the performance of the prefetch based code. This means, if we can reduce the number of harmful prefetches, we can improve performance significantly and make prefetching a scalable optimization. Clearly, making optimizations such as prefetching scalable is very important for current CMPs, and one can expect that it will be even more so for future many-core architectures (21). The rest of this paper presents and evaluates a novel data prefetching scheme for CMPs to achieve this goal.

[1] The details of our experimental setup and applications will be given later after discussing our proposed scheme.

[2] It needs to be mentioned that the degradation in performance of the prefetched code with larger core counts cannot solely be explained by the increase in the contribution of harmful prefetches. It is also known for example that, as the number of cores increases, the contribution of conflict misses due to normal reads may also play a role (47).

Before starting a discussion of the technical details of our scheme however, we want to briefly discuss why in general independent prefetching described and evaluated above faces performance problems when it is used in CMPs with large number of cores. First, prefetch instructions inserted by the compiler bring certain CPU overhead (whether or not prefetching itself is harmful), though one can expect this overhead not to be a major player. Second, as the number of cores increases, the number of prefetches issued for data that are already in the L2 cache increases. That is, multiple cores can issue prefetches for the same data. While as explained earlier our prefetch implementation filters some of such duplicate prefetches before they go to the cache, there is still an overhead associated with each such (filtered) prefetch request. In any case, eventually, some duplicate prefetches do go to the cache. Third, as the number of cores increases, chances for negative interferences between prefetch instructions and normal data fetches also increase. It is important to emphasize that reducing the number of harmful prefetches can help reduce the impacts of all these factors.

3. Data Prefetching for CMPs

3.1 Target CMP Abstraction

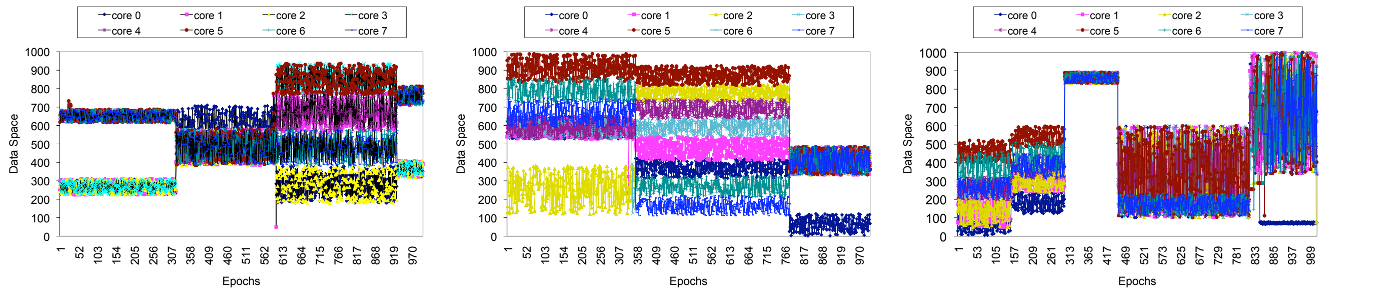
In our target CMP architecture, on-chip memory space is organized as a hierarchy (each core having a private L1 cache and all cores sharing the same on-chip L2 space). The coherence at the L1 level is assumed to be maintained using a MESI-like protocol. The prefetching algorithms we discuss and evaluate in this work target the shared on-chip L2 cache, that is, we are trying to optimize the performance of data prefetching from the off-chip memory to the shared L2 space. Although not essential for our proposed scheme, we also assume that only one thread is executed on each core at any given time. As a result, we use the terms "thread" and "core" interchangeably when there is no confusion.

3.2 A Quantitative Analysis of Data Access/Sharing Patterns

To further motivate our approach, we present data access patterns from three of our benchmarks (galgel, equake, and ammp) when 8 cores are used. In all these figures, the x-axis denotes the execution progress and the y-axis captures the addresses of the data elements accessed. In obtaining our graphs, the observed application execution period is divided into equal-sized epochs,[3] and the addresses of the accessed data elements are recorded. We see from Figure 4(a) that, this application's execution can be divided into four distinct phases. In the first phase, while cores 0, 1, 2 and 3 share the same set of elements, the rest of the cores share a lot of data between them. In the second phase, we see that all cores except core 0 share the same set of data elements. In the third phase, we can divide the cores into groups of two, the cores in each group sharing data between them. The last phase is very similar to the first one (i.e., cores can be divided into two groups, each having 4 cores) except that the set of elements accessed are different. The data access pattern illustrated in Figure 4(b) shows three easily identifiable phases. In the first phase, four of the cores share the same set of elements, whereas the remaining cores access a large region of the address space without much sharing. In the second phase, there is not much sharing at all. But, in the third phase, all cores except core 0 share the same set of data elements. Finally, one can also observe several distinct phases in the data access/sharing pattern plotted in Figure 4(c).

We argue that a data prefetching strategy can be tuned based on these data access/sharing patterns, instead of allowing each core to issue prefetch requests independently (as we discussed earlier independent prefetching did not generate good results with large number of cores). Consider for instance the patterns shown in Fig-

[3] In most of our applications the phases are repetitive, that is, the same data pattern occurs multiple times when the entire application execution period is considered. In these graph of Figure 4, we only look at a small portion of the execution.



(a) galgel (b) equake (c) ammp

Figure 4. Data access patterns exhibited by three applications executing using 8 cores. The x-axis captures the initial epochs of the application, and the y-axis represents the data elements accessed. All data elements are renumbered (sequentially) and the total data space is divided into 1000 equal-sized regions).

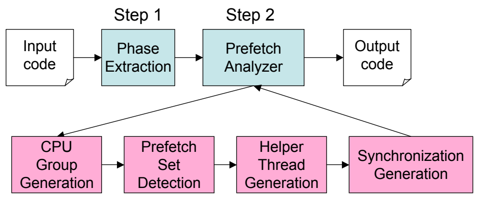
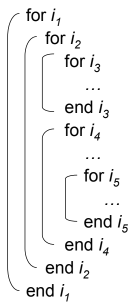


Figure 5. Major components of our helper thread based data prefetching scheme.



```
for i1
  for i2
    for i3
      …
    end i3
    for i4
      …
      for i5
        …
      end i5
    end i4
  end i2
end i1
```

Figure 6. An example that shows three different SNLNs.

ure 4(a). In the first phase of this application, it may be a good idea to have one prefetcher (helper) thread for cores 0, 1, 2, 3 and another prefetcher thread for cores 4, 5, 6, 7. In this case, the application threads running on these cores will not perform any data prefetching. Instead, all prefetches will be issued by the helper threads on behalf of them. In this way, the number of harmful prefetches and their impact on performance can be reduced significantly. Specifically, a helper thread will issue a single prefetch for each data element in a timely manner and consequently the number of times a prefetched data conflicts with other prefetches or with normal reads gets reduced.

Similarly, in the second phase of this application, all cores except core 0 can share the same helper thread that can prefetch for all of them, and core 0 can perform its own prefetches. In the third phase of this application, we can assign a prefetcher thread for each group of two. In the second phase in Figure 4(b), on the other hand, all cores can perform their own prefetches, as there is no sharing across their data accesses. Although the results in Figure 4 are all for an 8 core execution, we observed similar mixed data access patterns with other core counts as well. In the rest of this section, we present the technical details of our new data prefetching scheme that takes advantage of this diversity in data access and sharing patterns. Basically, our L2 aware data prefetching divides the program into *phases* and, for each phase, divides the cores into groups based on their data sharing patterns. Each group contains threads (cores) that share data among them, and is assigned a single helper thread that performs prefetching on behalf of all the cores in that group. This relieves the application threads from issuing prefetch requests and cuts the number of harmful prefetches.

3.3 Details of Our Approach

Our approach has two major steps, as illustrated in Figure 5. In the first step, called the *phase extraction,* the compiler analyses the multi-threaded application code and divides it (logically) into phases. The second step, called *prefetch analyzer*, operates on each phase independently (i.e., it handles one phase at a time). Its main job is to decide what prefetch instructions to insert in the code and where to insert them. Below, we discuss these steps in detail.

3.3.1 Step 1: Phase Extraction

In the first step, the compiler analyzes the application code and divides it into phases. There exist several static/dynamic phase extraction schemes proposed in literature (10; 3; 9; 42; 19; 2). In our approach, we use a simple static phase partitioning scheme based on the concept on Singly-Nested Loop Nests (SNLNs), proposed originally in (55). An SNLN is either a perfectly-nested loop nest or an imperfectly-nested loop nest where the imperfect code does not have any loops or complex control flow. In fact, in our implementation, imperfect code is allowed to have only assignment statements, call instructions, and simple conditionals. As an example, consider the code sketch given in Figure 6.

In this sketch, we have three different SNLNs. The first one includes loop i_3; the second one includes loops i_4 and i_5; and the last one includes the outer loops i_1 and i_2. Once the SNLNs are identified, our approach operates as an SNLN granularity, that is, each SNLN is assumed to be a phase and is processed independently. The outer SNLNs (e.g., the one that contains loops i_1 and i_2 in the example above) are not operated unless they include imperfectly nested non-loop statements for which data prefetching can be performed. In the example above, if there were assignment statements between loops i_2 and i_3, we would perform prefetching for them as well. Also, in operating on an SNLN, our approach targets parallel loops (in that SNLN); the serial loops are handled using the conventional prefetching strategy. In the rest of this section, we describe how our approach handles a given phase (SNLN). The reason that we adopt this particular phase extraction scheme is two-fold. First, we observed after experimenting with several multi-threaded codes that different SNLNs have typically very different data access and sharing patterns. Therefore, it makes sense to consider two different SNLNs as two different phases. Second, an SNLN is the largest code segment for which we know how to perform data prefetching. A similar observation is made in (55) for other types of optimizations (e.g., loop permutation, tiling) as well.

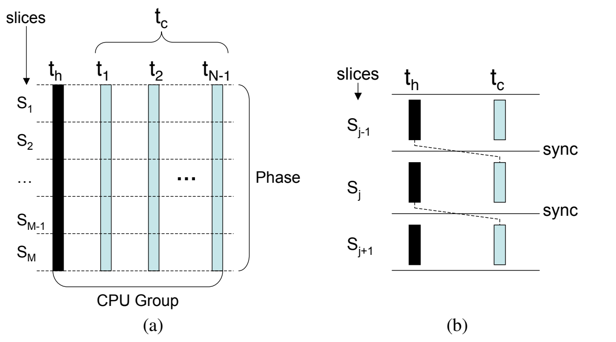


Figure 7. (a) Overview of how helper thread and computation threads are scheduled in a given phase. Note that a phase identified during compiler analysis (Step 1) contains multiple slices. (b) Synchronization strategy between helper thread (t_h) and computation thread (t_c).

3.3.2 Step 2: Prefetch Analyzer

As shown in Figure 5, this step has four subcomponents, namely, CPU group generation, prefetch set determination, helper thread generation, and synchronization generation. The goal of this second step is to restructure each identified program phase into *computation threads* and *helper threads*. In our approach, computation threads (also called *application threads*) perform computation only while the helper threads perform data prefetches on behalf of the computation threads. The details of each of these four subcomponents are discussed below.

CPU Group Generation The first task of the prefetch analyzer is to divide the cores into groups, which we call *CPU Groups.* Two processor cores (or more) are assigned to the same CPU group if they share a large number of data elements. Let $\mathcal{Y}_i$ denote the set of elements accessed by thread i in the phase (SNLN) being analyzed. If $\frac{\texttt{sizeof}(\mathcal{Y}_i)\ \cap\ \texttt{sizeof}(\mathcal{Y}_j)}{\texttt{sizeof}(\mathcal{Y}_i)\ +\ \texttt{sizeof}(\mathcal{Y}_j)} > \Delta$, threads i and j are assumed to belong to the same CPU group. Here, Δ is the *sharing density threshold* and can be set to different values to control groupings of the cores. Note that a large Δ value means that two cores are placed into the same CPU group only if they share a large fraction of data accesses between them, and in such cases, it is better to assign a dedicated prefetch thread (helper thread) to those cores, instead of allowing each core to perform its own data prefetching. For example, if Δ is set to 70%, cores that have a Δ of 70% or higher in a phase are assigned the same data prefetcher in that phase.

Prefetch Set Determination After determining CPU groups, the next task is to determine the set of data elements that need to be prefetched for each CPU group identified. Note that a program typically has multiple phases and each phase also has multiple CPU groups depending on the data sharing patterns among the cores. *A prefetch set is determined for each CPU group in each phase.* We start by dividing the phase into M *slices*, i.e., $S_1, S_2, \cdots, S_M$. More specifically, we divide the entire loop iterations in a phase into equal-sized chunks using predefined slice size, which is 10% of total number of iterations. Let us focus on a CPU group that holds N cores. We use $\mathcal{I}_{i,j}$ to denote the iterations assigned to computation thread t_i in slice j, where $1 \leq i \leq N-1$[4] and $1 \leq j \leq M$. We then compute $\mathcal{X}_{i,j}$, the data elements that are accessed by t_i in slice j as follows:

$$\mathcal{X}_{i,j} \quad = \quad \{\vec{d}\mid \exists\vec{I}\in\mathcal{I}_{i,j},\ \exists R\in\mathcal{R}_{i,j} \text{ such that } R(\vec{I})=\vec{d}\}.$$

In this equation, $\mathcal{R}_{i,j}$ is the set of references to the array data; R represents a reference in the loop nest (i.e., a mapping from the iteration space to the data space); $\vec{I}$ is an iteration point; and $\vec{d}$ is the index to data elements (i.e., array subscript function). For example, if two references $A[i_1][i_2]$ and $B[i_1][i_2]$ appear within a nest constructed using two loops i_1 and i_2, we have $\vec{I}=(i_1 \quad i_2)^T$, and $\mathcal{R}_{i,j}$ is $\{A[i_1][i_2], B[i_1][i_2]\}$. Since $\mathcal{I}_{i,j}$s in a particular slice are likely to share data elements, we can expect that:

$$\mathcal{X}_{x,j}\cap\mathcal{X}_{y,j}\neq\emptyset,\ \text{ for } x\neq y.$$

After calculating $\mathcal{X}_{i,j}$ for each thread t_i in the j^{th} slice, our next task is to figure out the total set of data elements accessed by the slice S_j, denoted as $\mathcal{X}_j$, which can be obtained by the union of $\mathcal{X}_{i,j}$ sets:

$$\mathcal{X}_j = \bigcup_{1\leq i\leq N-1} \mathcal{X}_{i,j},$$

where N is the number of CPU cores as stated earlier. Note that the size of $\mathcal{X}_j$ is far smaller than the total of each $\mathcal{X}_{i,j}$s if the sharing density between threads is high. That is, we have $\texttt{sizeof}(\mathcal{X}_j) \ll \sum_{i=1}^{N-1}\{\ \texttt{sizeof}(\mathcal{X}_{i,j})\ \}$ if the threads access the same (small) data region. Therefore, the higher the sharing density Δ (as defined earlier), the larger the gap between $\texttt{sizeof}(\mathcal{X}_j)$ and $\sum_{i=1}^{N-1}\{$ $\texttt{sizeof}(\mathcal{X}_{i,j})\}$. Note that, if there are multiple array data in slice M, we need to build a $\mathcal{X}_j$ set for each array for which data prefetching will be performed.

Helper Thread Generation The next step is to generate code for helper threads that contain data prefetch instructions. To accomplish this, both computation and helper thread code should have explicit boundaries in each slice. To obtain these boundaries, we apply strip-mining (56) to the computation threads and helper threads. Besides strip-mining, there are additional tasks to be performed for helper threads. The first task is to generate the addresses to be prefetched for the elements in each $\mathcal{X}_j$, and then insert prefetch instructions for these addresses. Since iterations in a slice can share data, we do not want to issue multiple prefetches for the same data block. Our solution to prevent redundant prefetches to the same data block is to use the Omega Library (39) to generate a loop (or a set of loops depending on the addresses to be generated) that enumerates the addresses of the elements in a $\mathcal{X}_j$. After this, these individual loops for different slices are combined to generate a compact code where the outer loop iterates over the different slices (S_j) and the inner loop iterates over the addresses of the data blocks to be prefetched in a given slice. The goal of this is to generate a compact code as much as possible. Note that while we use the Omega Library for address generation, any other polyhedral arithmetic tool could also be used for this purpose. After completing these steps, we have the codes for both computation threads and helper thread. The code for the helper thread also contains data prefetch instructions, and the code for both threads, computation and helper, are generated such that they have explicit slice boundaries. This explicit boundaries are used for inserting synchronizations between the computation and helper threads, which is discussed in the following paragraph.

Synchronization Generation The last task of our compiler algorithm is to generate synchronizations between the helper thread and computation threads to ensure timely pipelining of data prefetches. Figure 7(b) illustrates the interaction between the computation threads and the helper thread, the latter of which prefetches on behalf of computation threads. As explained earlier, the phase is divided into slices, each of which contains $\mathcal{I}_j$ iterations. As shown in Figure 7(a), the synchronizations occur in these slice boundaries in order to make sure that when the jth computation threads start execution, all the prefetches that bring $\mathcal{X}_j$ to the shared L2 cache should be finished. That is, the computation on that data element cannot start until all the data elements in $\mathcal{X}_j$ are brought into the shared cache. This is achieved in our scheme by inserting explicit synchronization instructions, `sync()`, between the helper thread and computation threads. As shown in Figure 7(b), the prefetch instructions to the data in $\mathcal{X}_j$ is issued at the beginning of slice $(j-1)$. Issuing the prefetch instructions for the data in $\mathcal{X}_j$, the helper thread

[4] Without loss of generality, we assume that cores 1 through $N-1$ execute application threads (one thread per core) and the Nth core executes the helper thread.

```
Input: $P = (L_1, L_2, \cdots, L_Q)$, where $Q$ is the number of phases in $P$;
Output: $P' = (L'_1, L'_2, \cdots, L'_Q)$;
$\mathcal{C} = (C_1, C_2, \cdots, C_D)$ – CPU core groups that belong to a particular phase;
$C_l$ – CPU core group that exhibits accesses on shared data;
$|C_l|$ – the size of $C_l$;
$T$ – minimum CPU group size, default is 2;
$S$ – number of iterations used for strip-mining the original loop nest;
$\Delta$ – sharing density threshold;
$\mathcal{Y}_i$ – the set of data elements accessed by thread $i$;
for each $L_i \in P$ do
    detect singly-nested loop nest (SNLN) and mark it as a phase;
end for
for each detected phase $\in P$ do
    for each thread, $t_x$ and $t_y \in$ phase do
        if $\frac{\texttt{sizeof}(\mathcal{Y}_i) \ \cap \ \texttt{sizeof}(\mathcal{Y}_j)}{\texttt{sizeof}(\mathcal{Y}_i) \ + \ \texttt{sizeof}(\mathcal{Y}_j)} > \Delta$ then
            $t_i$ and $t_j$ belong to the same $C_l$;
        end if
    end for
    for each $C_l$, $C_l \in \mathcal{C}$ do
        if $|C_l| \leq T$ then
            for each computation thread $t_i \in C_l$ do
                apply normal data prefetching scheme;
            end for
        else
            duplicate the original $t_i$ and mark it as helper thread;
            let $N$ be the size of $C_l$, i.e., $|C_l| = N$;
            allocate $N - 1$ cores to the $t_i$;
            allocate one core to the duplicated helper thread;
            recalculate the loop bounds for $t_i$;
            strip-mine both the main and helper thread using $S$;
            for each thread $t_i$, $t_i \in C_l$ do
                compute $\mathcal{X}_{i,j}$ for the $j$th slice;
            end for
            compute $\mathcal{X}_j$;
            call Omega_library to enumerate iterations of each $\mathcal{X}_j$;
            for each array $\in L_x$ do
                add ``prefetch()'' for $\mathcal{X}_j$ in $t_h$;
            end for
            add ``sync()'' for both $t_c$ and $t_h$;
        end if
    end for
end for
```

Figure 8. Compiler algorithm for transforming a given code to insert explicit prefetch instructions and necessary synchronizations.

needs to synchronize with computation threads, which is to ensure that all the computations in slice $j - 1$ are completed before proceeding to slice j. After completion of all synchronizations, the helper thread resumes prefetching for the data in $\mathcal{X}_{j+1}$ while the computation threads start executing their computation on the data in $\mathcal{X}_j$.

Figure 8 gives a high-level view of our compiler algorithm explained so far. It takes an input program (P) divided into Q phases, along with the number of cores (N) in each identified CPU group, and generates the two versions of the code: helper thread and computation threads.

3.3.3 Discussion

This section briefly discusses a couple of important points regarding our scheme. The first issue we want to discuss is about helper threads. As explained so far, our approach generates a helper thread for each CPU group. Therefore, for small sized CPU groups, we might hurt performance due to reduced number of CPU cores. We tried to address this issue as follows. The first option we tried was to execute the helper thread in a CPU group using one of the cores within that group. That is one core is responsible for executing both computation thread and the prefetching code. The experimental results with this approach were not good and, in fact, close to those obtained through independent data prefetching.[5] The second option we tried was to stick to independent data prefetching when the size of a CPU group is smaller than a certain threshold value, which is preset as a parameter. As a matter of fact, we found that when the group size is two, independent data prefetching performs better rather than executing one computation thread and one helper thread separately. However, when the group size is three, our approach, which allocates two cores to computation and one dedicated core to data prefetching, performs better. Therefore, we set the minimum size for our approach to be applied to three throughout our experiments.

Note that the helper threads in our approach execute on separate cores and therefore they run in parallel with the application threads. Therefore, most of their execution latency is hidden in parallel execution except for synchronizations. Synchronization between helper threads and application threads occurs only at slice boundaries and thus they do not affect performance significantly. Note also that typically we have very few helper threads executing on the system at any given time (in fact, in about 75% of our phases, we had either 2 or 3 helper threads). In addition, a helper thread prefetches only the data requested by the application threads to which it is assigned, and brings a data element only once. In any case, *all the overheads associated with creating and managing helper threads are captured during our full system simulation and included in all our results.* We also want to mention that the code generated for helper thread and the memory footprint it takes up during execution is small enough to fit them into the L1 cache in our default configuration, which has 32KB L1 instruction cache and 32KB L1 data cache. This is because the helper thread has less computation to execute than computation threads. Therefore, the interference to shared L2 cache due to the helper thread is also negligible.

A limitation of our proposed scheme is that it is applicable to codes that can be analyzed by the compiler. It is important to note however that, while there are codes that fall outside of this category, many application codes from scientific computing domain and embedded image/video processing are amenable to compiler analysis and thus can be optimized using our prefetching scheme. The main point we want to make is that a simple extension of conventional compiler-directed prefetching (i.e., independent prefetching) does not work well when ported to a CMP environment, due to the negative interactions on the shared L2 space. Therefore, to obtain comparable benefits from compiler-directed prefetching in single core as well as multiple core environments, we need to modify the multi-core version to make it aware of data sharings across processor cores, as shown in this paper. It is to be noted that scientific applications and embedded image/video codes are maybe the first set of applications that can take advantage of future many-core architectures due to inherent parallelism they contain. Therefore, it is important to enhance their CMP performance using compiler techniques that are scalable, and our scheme is an effort in this direction.

Nevertheless, we also would like to briefly mention how our scheme can be made to work with other types of applications. Recall that the first step of our scheme is to identify the program phases. While this step cannot be done very easily for applications whose access patterns cannot be analyzed statically by the compiler, we might be able to use *profile information* to compensate for this. Specifically, we can first profile the application code and identify the data sharing patterns among the different cores. The outcome of this step can be shown in the form of plots, as illustrated in Figure 4, which help us identify the CPU groups as well as the number and types of the helper to use. Next, we can associate the identified data sharing patterns to code sections. Note that, while the first step gives us the CPU groups (i.e., which set of threads (cores) should be assigned a common I/O prefetching thread), the second step tells us the program segments where these groupings should be considered. After that, we can divide each phase into slices and place prefetch instructions in the code (helper threads)

[5] Note that this option is different from independent prefetching as all prefetches of a CPU group are performed by a single core which also executes a thread of the application. This is in contrast to independent prefetching where each core performs only its own prefetches.

```
/* 1st loop nest */                      /* 2nd loop nest */
for (i=0; i < N1; i++) {                 for (i=0; i < N1; i++) {
   for (j=0; j < N2; j++) {                 for (j=0; j < N2; j++) {
      a[i][j] += b[i][j];                      k = (int)(2j/N1);
}                                              c[i][j] *= d[k][j];
                                         }
```

Figure 9. Original code fragment with two loop nests.

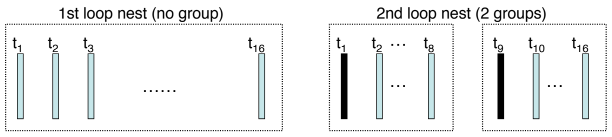


Figure 10. Computation and helper thread assignments in different loop nests.

that prefetch the required data elements for each slice (this set of elements can be determined using profile information). While this extension depends heavily on profile data, we believe it can be a first step toward handling application codes that are difficult to analyze statically.

Our approach is a compiler based prefetching mechanism that uses explicit prefetch instructions. In our modeled system, we did not have a hardware prefetcher. Certainly, existence of a hardware prefetcher can affect the behavior of our scheme (e.g., can increase the number of harmful prefetches). If a hardware prefetcher exists in the architecture, one way of using our approach would be disabling hardware prefetcher if the compiler can analyze the code and is able to insert prefetch instructions. If not, we can let the hardware prefetcher to take control. In this way, the execution can switch between two prefetchers at runtime.

3.4 Example

This section illustrates how our approach explained so far operates in practice using the example code fragment shown in Figure 9. For ease of illustration, let us assume that the code is parallelized using 16 cores. This code fragment contains computations that access four data arrays, a, b, c and d, using different references. As a result of our phase identification (see Step 1 in Figure 5), we have two distinct phases, i.e., each phase corresponds to a loop nest. The next step of our scheme is to extract CPU groups within a phase. Since each thread in the first loop nest accesses array elements independently, the first loop nest does not have any shared data accesses. The second loop nest on the other hand has two separate CPU groups where the iterations assigned to each group access their own shared data elements from array d.

Figure 10 gives a pictorial view of the thread distribution after applying our scheme to the two loop nests. When the first loop nest is in execution (there is no data sharing and hence no grouping), all threads (t_1 to t_{16}) are computation threads performing their own data prefetching (similar to (37)). For the second loop nest on the other hand, since there are two identifiable groups, our approach assigns a separate helper thread to each group (t_1 for the first group, t_9 for the second group). In this case, the application threads (t_2 through t_8 in the first group and t_{10} through t_{16} in the second group) do not perform any prefetches.

Figure 11(a) shows the code fragment after applying the conventional data prefetching scheme for core 1. This is because in the first loop nest (as stated earlier), there is no data sharing between cores, i.e., no grouping. Since the prefetches are executed in a block-size granularity, the outermost loop is modified to work with the specified block size, D. The innermost loop, on the other hand, iterates over the elements within a block. The code fragments for the remaining 15 cores have the same code structures except for the lower and upper loop bounds specific to each core.

For the second loop nest, our algorithm, after identifying the data sharing patterns, assigns a helper thread to each of the two

```
for (i=0, j=0; i < N1/16; i++) {
   prefetch(&b[i][j], D);
   for (jj=0; jj < ⌊N1/D⌋; jj+=D) {
      prefetch(&b[i][jj + D], D);
      for (j=jj; j < j + D; j++)
         a[i][j] += b[i][j];
   }
   for (j=⌊N1/D⌋ * D; j < N1; j++)
      a[i][j] += b[i][j];
}
```

(a) Loop Nest 1, core 1

```
for (i=0; i < N1/16; i++) {
   for (j=0; j < ⌊N1/D⌋; j++ {
      k = (int)(2(i + D)/N1);
      prefetch(&d[k][j], D);
      sync (syncvar);
   }
}
```

(b) Loop Nest 2, core 1 (helper thread)

```
for (i=0; i < N1/16; i++) {
   for (jj=0; jj < ⌊N1/D⌋; jj++ {
      for (j=jj; j < jj + D; j++) {
         k = (int)(2j/N1);
         c[i][j] *= d[k][j];
      }
      sync(syncvar)
   }
   for (j=⌊N1/D⌋ * D; j < N1; j++) {
      k = (int)(2j/N1);
      c[i][j] *= d[k][j];
   }
}
```

(c) Loop Nest 2, core 2 (computation thread)

Figure 11. Example application of our scheme.

Table 1. Experimental setup.

Processor Cores	8 processors with private L1 data and instruction caches, each processor is single threaded
Processor Model	Each processor is a 4-way fetch and issue in-order processor, 1 GHz frequency
Private L1 D-Caches	Direct mapped, 32KB, 64 bytes line size, 3 cycle access latency
Private L1 I-Caches	Direct mapped, 32KB, 64 bytes line size, 3 cycle access latency
Shared L2 Cache	8-way set associative, 8MB, 64 bytes line size, 15 cycle access latency
Memory	4GB, 200 cycle off-chip access latency
Sharing Density Threshold	70%
Slice Size	10%

groups. Since this takes away 2 cores (recall that we assign one thread per core), the iterations are redistributed (parallelism is retuned) among the remaining 7 threads in each group (see Figures 11(b) and (c)). The helper thread for the second loop nest has a single prefetch instruction for the shared data and a loop that contains instructions to prefetch the unshared data.

4. Experimental Evaluation

4.1 Setup

Our L2 aware prefetching scheme has been implemented using Phoenix, which is a framework from Microsoft for developing compilers as well as program analysis, testing and optimization tools (36). Phoenix defines an Intermediate Representation (IR) for programs, using ASTs, flow graphs, and an exception handling model. Its structure is highly modular which allows any of its components to be replaced with others without affecting the rest of the system. We also implemented the conventional prefetching scheme in (37) using Phoenix so that it can be compared to our approach.

We used SIMICS (34) to evaluate both our prefetching scheme and the conventional prefetching scheme. SIMICS is a full-system simulator used to run unchanged production binaries of the target hardware at high speeds. It can simulate a variety of architectures and operating systems. We enhanced the base SIMICS infrastructure with accurate timing models. We want to mention that this accurate timing models using the full system simulator which includes all effects due to operating systems allows us to capture real execution to the extent possible. Table 1 gives the default system configuration used in most of our simulations. We also changed the default values of some of our simulation parameters to conduct a sensitivity study.

We also added new synchronization APIs in our simulation platform. We assume that each synchronization API incurs a certain amount of latency, which is similar to the one used in Jung et al (22). For example, latency for transferring an address from the

Table 2. Important characteristic of the SPEC OMP benchmarks. The numbers are for the no-prefetch case except the last column which shows the percentage improvements in L2 misses by our approach with 12 cores.

Benchmark	L1 miss rates	L2 miss rates	Memory foot-print (MB)	L2 miss rate improvement
wupwise	1.6%	11.2%	1480	23.6%
swim	8.1%	20.6%	1580	26.1%
mgrid	11.3%	16.4%	450	28.1%
applu	0.8%	11.9%	1510	32.8%
galgel	1.1%	10.1%	955	19.3%
equake	2.4%	7.8%	860	26.0%
apsi	8.4%	7.2%	1650	19.6%
gafort	6.8%	11.6%	1680	15.7%
fma3d	4.1%	7.6%	1020	23.2%
art	0.9%	38.0%	2760	34.3%
ammp	0.6%	5.3%	475	16.8%

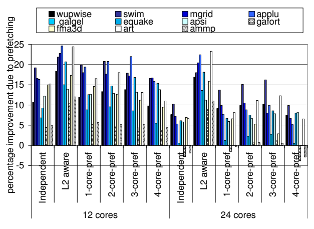


Figure 13. Comparison of different data prefetching schemes (savings are with respect to the no-prefetch case).

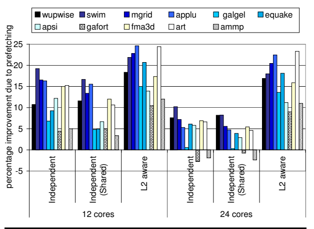


Figure 14. Comparison with alternate independent prefetching scheme (savings are with respect to the no-prefetch case).

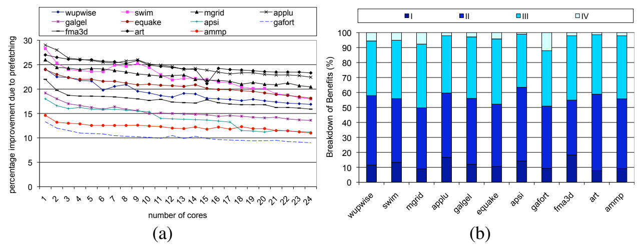


Figure 12. (a) Performance improvement (over the no-prefetch case) when our data prefetching scheme is used. (b) Breakdown of the benefits brought by our scheme over the conventional prefetching: (I) benefits due to reducing the CPU cycles spent by application threads while executing prefetch instructions; (II) benefits due to coordinating accesses from different threads to shared data; (III) benefits due to reducing the number of duplicated prefetches; (IV) benefits that we could not include into one of these three categories.

application to a synchronization register in the memory controller is about 68 cycles. The effect of these instructions is already captured in our results.

Table 2 gives important characteristics of the application codes used in this study. Basically, we used the entire SPEC OMP benchmark suite (45), which includes applications to evaluate the performance on OpenMP based implementations (we used the latest version which is V3.2). All the performance data, cache miss rates for L1 and L2, and memory footprints, presented in Table 2 are for the case when no data prefetching is used (i.e., the no-prefetch case). The last column in Table 2 shows the percentage improvements in L2 misses with our approach when 12 cores are used. All the applications use the reference input set and are fast forwarded to the beginning of the main loops. We warm the caches for about 1 billion instructions and collect results until the loop iterations terminate.

4.2 Results

Recall that Figure 3(a) gives the performance improvements the independent prefetching (with the per core FIFO implementation) brings over the no-prefetch case, and Figure 3(b) presents the contribution of harmful prefetches. As discussed earlier in Section 2, the independent prefetching fails to scale as we increase the number of cores in the CMP. In fact, in some applications, it even generates worse results than the base case which does not use any data prefetching.

Figure 12(a) gives the performance of the prefetched code when our scheme is used, instead of conventional prefetching. As in Figure 3(a), the improvements are with respect to the base case which does not employ any data prefetching (i.e., with respect to the no-prefetch case). When Figure 12(a) and Figure 3(a) are compared, it can be seen that our prefetching scheme performs much better than the conventional prefetching scheme (note that the improvements in both Figures 3(a) and 12(a) are with respect to the no-prefetch case). In fact, we observe that the percentage benefits from prefetching are quite stable across varying core counts when our L2 aware prefetching scheme is used. To explain the difference between Figure 12(a) and Figure 3(a), we also collected statistics that help us understand where these improvements are coming from. Figure 12(b) plots the breakdown of benefits brought by our prefetching approach, when averaged over all core counts used. We divide the benefits of our new data prefetching scheme into four categories: (I) Benefits due to reducing the CPU cycles spent by application threads due to prefetch instructions inserted by the compiler. Since our scheme reduces the number of prefetches issued, it also reduces the CPU overhead due to fetching, decoding and committing the prefetch instructions. (II) Benefits due to coordinating accesses from different threads to shared (prefetched) data. As explained earlier, our approach restructures a given phase such that the cores perform coordinated accesses to the shared data (on a slice basis). As a result, our approach in a sense improves locality of the prefetched data in the L2 cache. More specifically, the first set of shared data elements are brought into the L2 cache, and all the cores operate on it. Then, they synchronize (with the helper thread) and the second set of shared data are brought and operated on, and so on. In this way, our approach increases the odds that shared data will be caught in the cache when it is reused. (III) Benefits due to reducing the number of harmful prefetches. As explained earlier, our approach reduces the number of duplicate prefetches through the use of a single helper thread per CPU group. (IV) Benefits that we could not include into one of these three categories. The results in Figure 12(b) clearly show that most of the benefits we obtain come from eliminating duplicate prefetches and reducing negative interactions on the shared cache, though the exact contribution of each category varies from benchmark to benchmark.

We now compare the performance of our prefetching algorithms against a simpler prefetching scheme, which allocates a fixed number of helper threads to perform prefetching. We use "x-core-pref" to denote a scheme where x cores are devoted for prefetching on behalf of the others throughout the entire execution period (in each phase) and the remaining cores are used for application execution. We present the results with $1 \leq x \leq 4$, as higher x values always generated worse results than those reported in here. Our first observation from Figure 13, which presents results for the 12 and 24 core cases, is that 1-core-pref, 2-core-pref and 3-core-pref generally result in better performance than the independent prefetching scheme. However, the same cannot be said for 4-core-pref, as in this case too many (4) cores allocated for just prefetching and this hurts performance, especially when the number of cores is 12. The second observation one can make from Figure 13 is that our L2 aware prefetching scheme outperforms all the alternate prefetching schemes tested for all application codes we have, which indicates that fixing the number of prefetch threads may not be the best option in general.

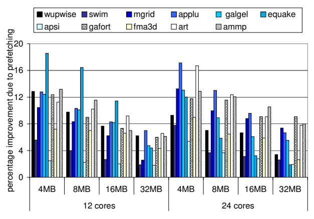


Figure 15. Impact of different cache sizes when 12 and 24 cores are used (savings are with respect to the independent prefetching case).

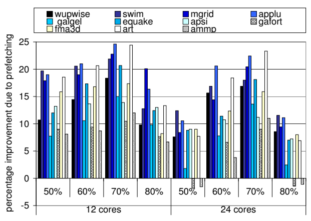


Figure 16. Impact of variation on the sharing density threshold (savings are with respect to the no-prefetch case).

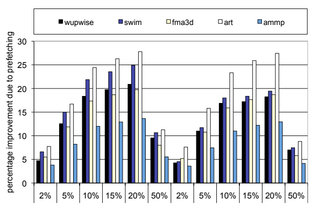


Figure 17. Impact of different slice sizes (savings are with respect to the no-prefetch case).

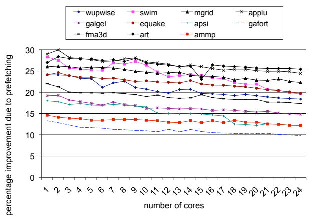


Figure 18. Comparison with optimal prefetching scheme.

As stated earlier, we also implemented an alternate version of the independent prefetching scheme where the runtime system applies duplicate prefetching check across all cores. That is, if a core issued prefetch request for a data item, another prefetch request for the same data item from a different core can be caught and filtered in this alternate implementation. As discussed earlier, this implementation is more costly than our default implementation, which applies duplicate checks to only to the prefetches issued by the same core. Figure 14 presents the percentage performance improvements with this new implementation (denoted as independent (shared)). The results with the default independent prefetching implementation and our L2 aware scheme are also reproduced for ease of comparison. We observe that, while there are exceptions, in general the per chip version of the prefetch filtering generates worse results than the default implementation (per core based) due to increased overhead incurred within the runtime system. However, we want to emphasize that even if we could implement a hypothetical (independent prefetching) scheme that eliminates all duplicate prefetches without incurring any costs, that version would not be better than our L2 aware scheme. This is because, as discussed above, a large portion of the benefits of our approach comes from improved L2 locality (category (II) in Figure 12(b)), which cannot be achieved by an independent prefetching strategy.

In the rest of this section, we present results from our sensitivity experiments. In each sensitivity experiment, we change the value of one simulation parameter only; the values of the remaining parameters remain as shown in Table 1. The first parameter we study is the L2 cache capacity. Recall that the default L2 capacity used in our simulations so far is 8MB. The results (savings over the independent prefetching case) with different cache capacities are given in Figure 15 for the 12 and 24 core cases. While, as expected, the achieved savings get reduced as we increase the L2 cache capacity, even with the largest L2 capacity used (32MB), we achieve on average 4.7% improvement for the 12 core case and 5.2% improvement for the 24 core case.

Recall that so far in our experiments we assigned a separate helper thread to two or more threads if the sharing density is 70% or higher (in other words, the sharing density threshold was 70%). Figure 16 shows the percentage improvement results when the sharing density threshold is varied between 50% and 80%. Our first observation is that when we set the threshold to 80%, the savings are not good. The main reason is that, with such a high threshold, the compiler cannot find much opportunity to apply our optimization, and most of the time, each core ends up with performing its own data prefetching. On the other hand, when the threshold is low (50%), our approach starts behaving similar to the independent prefetching case. Next, we present the results with different slice sizes (S) in Figure 17. In our default setting, the slice size is set to 10% of the total loop iteration count. We see from these results that, while the slice size has some impact on our results, unless one works with too small or too large sizes, the results obtained with different values of S are not too far from each other. We also would like to mention that while in this section we presented a sensitivity analysis of our approach to sharing density threshold and slice size, we believe both these parameters are important and require a thorough study. In particular, it is an open research problem whether appropriate values for these parameters can be automatically derived by the compiler, or whether profile information can be used for this purpose. Exploring this issue further is in our future research agenda.

Finally, we also performed experiments with a (hypothetical) optimal prefetching scheme that eliminates all harmful prefetches without any cost and maximizes data reuse in L2. To obtain this, we augmented our base SIMICS simulation platform and collected traces for cache accesses. These collected traces have perfect knowledge on future access behavior and we used them for simulating optimal prefetching such that there exist no prefetches that leads to a harmful prefetch. The results with this hypothetical scheme are presented in Figure 18. If we compare these results against those obtained using our L2 aware prefetching scheme (Figure 12(a)), we see that our savings come very close to those that can be achieved using the optimal scheme (the average difference is about 7%), indicating that our approach performs very well in practice (i.e., it eliminates most of the harmful prefetches that can be eliminated and maximizes locality of the shared L2 data as much as possible).

5. Related Work

Data prefetching is known to be a very effective way of improving performance of applications by hiding CPU stall time by issuing instructions ahead of the time when they are actually needed (37; 12; 33; 49; 33; 6; 8; 16; 24; 29; 32; 43; 46; 51; 53; 52; 1; 7; 40; 41; 14). Since prefetched blocks can pollute the normal cached blocks, prefetching should be designed and implemented carefully. Mowry et al (37) used compiler-guided information to minimize any unnecessary or redundant prefetches. They also employed coordination with runtime layer in order to eliminate the total message traffic with prefetching for multiprocessors (32). Vander Wiel and Lilja also used compiler help for data prefetching controller (51).

In the context of SMT (simultaneous multi-threading) and CMP (chip multiprocessor) domain, several prior studies considered employing a helper thread to hide memory latency (22; 30; 46; 25; 28; 31). This is done by converting an extracted program slice that executes critical instructions into helper thread. Jung et al (22) use a helper thread based prefetching scheme for loosely-coupled processors, like the modern CMPs. Kim et al (25) employ a similar scheme that employs helper threads running in spare hardware contexts ahead of the main computation to start the memory operations early so that memory latency can be tolerated. Liao et al (28) pinpoint the trigger points in the executable and generate a new executable with the prefetch threads embedded. Song et al (44) proposed a detailed compiler framework that generates helper thread prefetching for dual-core SPARC microprocessors. Our approach is different from these efforts in two aspects. First, we consider the cases where there are multiple CPU cores, and therefore, our scheme applies a different code modification, which is slice based. To our knowledge, the prior efforts based helper thread target at two-core cases. That is they runs one one application in one core and the prefetcher in the other. In contrast, our scheme is more general and can work with any number of cores. In addition, to our knowledge, none of the prior studies considered a program phase based approach where a number of helper threads that perform data prefetching on behalf of application threads changes during execution time. Second, we target array-intensive multi-threaded applications.

Prefetching is also extensively studied in the context of improving I/O performance (50; 38; 4; 13). Mowry et al use compiler-guided information to manage I/O prefetch commands effectively (38). They also studied the cases where multiple processes issue I/O prefetch commands concurrently (4). Li and Shen (27) proposed a memory management framework that handles non-accessed but prefetched pages separately from the rest of the memory buffer cache. Recent studies to improve conventional I/O prefetching using additional file and access history information include Diskseen (11), Competitive Prefetching (5) and AMP (15). In comparison to these studies, our work targets multiple prefetching on a CMP.

6. Conclusion and Future Work

Shared on-chip cache management is a crucial CMP design aspect for the performance of the system. It is very important to maximize the effective utilization of this cache through hardware and software based techniques. While compiler based data prefetching is known to be an effective technique for improving data cache behavior, it has not been tested thoroughly in the context of CMPs. This paper first presents such an evaluation using the entire suite of the SPEC OMP applications, and then proposes a new prefetching scheme for the shared L2 based CMPs. The proposed prefetching scheme is oriented toward reducing the number of harmful prefetches. The collected results indicate that, in 12 and 24 core cases, our scheme improves overall parallel execution latency by 18.3% and 16.4% respectively over the no-prefetch case and by 6.4% and 11.7% respectively over the independent prefetching case (on average). Our results also show that the proposed prefetching scheme is robust across a wide range of values of our major simulation parameters such as the number of cores, on-chip cache capacity, sharing density threshold, and slice sizes. In our future work, we will focus on sharing density threshold and slice size parameters and explore whether compiler can help determine suitable values for them automatically. Work is also underway in integrating the proposed prefetching scheme with existing cache optimizations such as tiling, targeting a CMP setting.

References

[1] A. R. Alameldeen and D. A. Wood. Interactions Between Compression and Prefetching in Chip Multiprocessors. In *HPCA*, pages 228–239, 2007.

[2] Bala et al. Dynamo: a transparent dynamic optimization system. In *PLDI*, pages 1–12, 2000.

[3] Balasubramonian et al. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, pages 245–257, 2000.

[4] A. D. Brown and T. C. Mowry. Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently. In *OSDI*, pages 31–44, 2000.

[5] C. Li et al. Competitive Prefetching for Concurrent Sequential I/O. In *EuroSys*, pages 189–202, 2007.

[6] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. In *ISCA*, pages 223–232, 1994.

[7] Cooksey et al. A stateless, content-directed data prefetching mechanism. In *ASPLOS*, pages 279–290, 2002.

[8] Dahlgren et al. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *ICPP*, pages 56–63, 1993.

[9] A. S. Dhodapkar and J. E. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *ISCA*, pages 233–244, 2002.

[10] A. S. Dhodapkar and J. E. Smith. Comparing Program Phase Detection Techniques. In *MICRO*, pages 217–227, 2003.

[11] Ding et al. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *USENIX*, pages 261–274, 2007.

[12] Doshi et al. Optimizing Software Data Prefetches with Rotating Registers. In *PACT*, pages 257–267, 2001.

[13] P. et al. Informed Prefetching and Caching. In *SOSP*, pages 79–95, 1995.

[14] I. Ganusov and M. Burtscher. Efficient Emulation of Hardware Prefetchers via Event-Driven Helper Threading. In *PACT*, pages 144–153, 2006.

[15] B. S. Gill and L. A. D. Bathen. AMP: Adaptive Multi-Stream Prefetching in a Shared Cache. In *USENIX FAST*, pages 185–198, 2007.

[16] E. H. Gornish and A. Veidenbaum. An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. *Int. J. Parallel Program.*, 27(1):35–70, 1999.

[17] Hammond et al. A Single-Chip Multiprocessor. *Computer*, 30(9):79–85, 1997.

[18] Hsu et al. Exploring the cache design space for large scale CMPs. *SIGARCH Comput. Archit. News*, 33(4):24–33, 2005.

[19] Huang et al. Positional Adaptation of Processors: Application to Energy Reduction. In *ISCA*, pages 157–168, 2003.

[20] Intel. Intel Core Duo Processor and Intel Core Solo Processor on 65 nm Process, January 2007. Datasheet.

[21] Intel Corporation. Intel Develops Tera-Scale Research Chips, 2006. `http://www.intel.com/pressroom/archive/releases/20060926corp_b.htm`.

[22] Jung et al. Helper Thread Prefetching for Loosely-Coupled Multiprocessor Systems. In *IPDPS*, 2006.

[23] Kalla et al. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, 2004.

[24] A. Ki and A. E. Knowles. Adaptive data prefetching using cache information. In *ICS*, pages 204–212, 1997.

[25] D. Kim and D. Yeung. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *ASPLOS*, pages 159–170, 2002.

[26] Kongetira et al. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[27] C. Li and K. Shen. Managing prefetch memory for data-intensive online servers. In *USENIX FAST*, pages 253–266, 2005.

[28] Liao et al. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In *PLDI*, pages 117–128, 2002.

[29] Lu et al. The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System. In *MICRO*, page 180, 2003.

[30] Lu et al. Dynamic Helper Threaded Prefetching on the Sun UltraSPARC CMP Processor. In *MICRO*, pages 93–104, 2005.

[31] C.-K. Luk. Tolerating Memory Latency through Software-controlled pre-execution in Simultaneous Multithreading Processors. In *ISCA*, pages 40–51, 2001.

[32] C.-K. Luk and T. C. Mowry. Architectural and compiler support for effective instruction prefetching: a cooperative approach. *ACM Trans. Comput. Syst.*, 19(1):71–109, 2001.

[33] Luk et al. Profile-guided post-link stride prefetching. In *ICS*, pages 167–178, 2002.

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

[35] C. McNairy and R. Bhatia. Montecito - The next product in the Itanium(R) Processor Family, 2004. In Hot Chips 16, `http://www.hotchips.org/archives/`.

[36] Microsoft. Phoenix as a Tool in Research and Instruction. `http://research.microsoft.com/phoenix/`.

[37] Mowry et al. Design and Evaluation of a Compiler Algorithm for Prefetching. In *OSDI*, pages 62–73, 1992.

[38] Mowry et al. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *OSDI*, pages 3–17, 1996.

[39] W. Pugh and D. Wonnacott. Going Beyond Integer Programming with the Omega Test to Eliminate False Data Dependences. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):204–211, 1995.

[40] Rabbah et al. Compiler orchestrated prefetching via speculation and predication. In *ASPLOS*, pages 189–198, 2004.

[41] Roth et al. Dependance Based Prefetching for Linked Data Structures. In *ASPLOS*, pages 115–126, 1998.

[42] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *ISCA*, pages 336–349, 2003.

[43] Shi et al. Coterminous locality and coterminous group data prefetching on chip-multiprocessors. In *IPDPS*, 2006.

[44] Song et al. Design and Implementation of a Compiler Framework for Helper Threading on Multi-Core Processors. In *PACT*, 2005.

[45] SPEC. SPEC OMP Version 3.0 Documentation (OpenMP Benchmark Suite). `http://www.spec.org/omp/`.

[46] Spracklen et al. Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications. In *HPCA*, pages 225–236, 2005.

[47] Srikantaiah et al. Adaptive set pinning: managing shared caches in chip multiprocessors. In *ASPLOS*, pages 135–144, 2008.

[48] Sun Microsystems. UltraSPARC-II Enhancements: Support for Software Controlled Prefetch, 1997. White Paper WPR-0002.

[49] Tian et al. Impact of Compiler-based Data-Prefetching Techniques on SPEC OMP Application Performance. In *IPDPS*, page 53.1, 2005.

[50] Tomkins et al. Informed Multi-Process Prefetching and Caching. In *SIGMETRICS*, pages 100–114, 1997.

[51] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, 2000.

[52] Wang et al. Guided Region Prefetching: A Cooperative Hardware/Software Approach. In *ISCA*, pages 388–398, 2003.

[53] S. P. V. Wiel and D. J. Lilja. A compiler-assisted data prefetch controller. In *ICCD*, pages 372–377, 1999.

[54] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *PLDI*, pages 30–44, 1991.

[55] Wolf et al. Combining Loop Transformations Considering Caches and Scheduling. In *MICRO*, pages 274–286, 1996.

[56] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 1995.