Profiler and Compiler Assisted Adaptive I/O Prefetching for Shared Storage Caches*

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

Sai Prashanth Muralidhara
Pennsylvania State University
smuralid@cse.psu.edu

Ozcan Ozturk
Bilkent University
ozturk@cs.bilkent.edu.tr

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

Ibrahim Kolcu
University of Manchester
ikolcu@umist.ac.uk

Mustafa Karakoy
Imperial College
mtk2@psu.edu

ABSTRACT

I/O prefetching has been employed in the past as one of the mechanisms to hide large disk latencies. However, I/O prefetching in parallel applications is problematic when multiple CPUs share the same set of disks due to the possibility that prefetches from different CPUs can interact on shared memory caches in the I/O nodes in complex and unpredictable ways. In this paper, we (i) quantify the impact of compiler-directed I/O prefetching – developed originally in the context of sequential execution – on shared caches at I/O nodes. The experimental data collected shows that while I/O prefetching brings benefits, its effectiveness reduces significantly as the number of CPUs is increased; (ii) identify inter-CPU misses due to harmful prefetches as one of the main sources for this reduction in performance with the increased number of CPUs; and (iii) propose and experimentally evaluate a profiler and compiler assisted adaptive I/O prefetching scheme targeting shared storage caches. The proposed scheme obtains inter-thread data sharing information using profiling and, based on the captured data sharing patterns, divides the threads into clusters and assigns a separate (customized) I/O prefetcher thread for each cluster. In our approach, the compiler generates the I/O prefetching threads automatically. We implemented this new I/O prefetching scheme using a compiler and the PVFS file system running on Linux, and the empirical data collected clearly underline the importance of adapting I/O prefetching based on program phases. Specifically, our proposed scheme improves performance, on average, by 19.9%, 11.9% and 10.3% over the cases without I/O prefetching, with independent I/O prefetching (each CPU is performing compiler-directed I/O prefetching independently), and with one CPU prefetching (one CPU is reserved for prefetching on behalf of others), respectively, when 8 CPUs are used.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers*; B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms

Algorithm, Experimentation, Performance

Keywords

Prefetching, Shared Storage Cache, Compiler, Profiler, Adaptive

1. INTRODUCTION

I/O prefetching is an important optimization for improving performance [27, 2, 33, 1, 36, 19, 11, 15, 41, 7, 34]. In I/O prefetching, data is brought from the disk to the memory cache (shared storage buffer) ahead of time to hide the latency of disk accesses. However, prefetching is known to be very sensitive to timing [33, 1]. First, an early prefetch may not be very useful as the data block brought into the memory cache can be discarded before it is used. Second, a late prefetch may not be very useful either since it cannot eliminate the entire disk access latency. Third, a prefetched data can be even "harmful" by kicking out a data block from the memory cache whose next usage is earlier than that of the prefetched block. In a shared storage cache (i.e., a memory cache in an I/O node shared by multiple CPUs), this type of "harmful prefetches" can involve different CPUs as well. For example, a prefetched data block can displace a data block which would be accessed earlier (by another CPU) than the prefetched data block, as illustrated in Figure 1(a). This type of harmful prefetches are referred to as "inter-CPU harmful prefetches," as opposed to "intra-CPU harmful prefetches," an example of which is given in Figure 1(b).

Clearly, the number of harmful prefetches can increase with the increased number of CPUs, and consequently, one can expect the problem to be more severe as the degree of sharing of an I/O node increases. This paper demonstrates the magnitude of this problem using four disk-intensive parallel applications and compiler directed I/O prefetching, and proposes a solution that employs code profiling and automated code restructuring. Our solution is based on several observations we made during our experiments:

- In general, the scalability of I/O-intensive applications (i.e., applications that frequently exercise disk subsystems of parallel machines) is not very good. As a result, a couple of CPUs can be used for purposes other than executing application threads, without impacting application performance too much.
- While compiler-directed I/O prefetching brings significant benefits in cases when a small number of CPUs (e.g., 1–4) are used, its performance degrades significantly as the number of CPUs is increased. Dramatic increases in harmful prefetches play a key role in this degradation with larger number of CPUs. This motivates for an approach that uses only a small set of CPUs to perform I/O prefetches, instead of all CPUs prefetching independently the data they need and competing over the shared storage cache.
- Data sharing patterns exhibited by an I/O-intensive application change significantly across the different phases of the application.

[*] This work is supported in part by NSF grants #0406340, #0444158, #0621402, #0724599, #0821527, and #0833126.

PACT'08, October 25–29, 2008, Toronto, Ontario, Canada.
Copyright 2008 ACM 978-1-60558-282-5/08/10 ...$5.00.

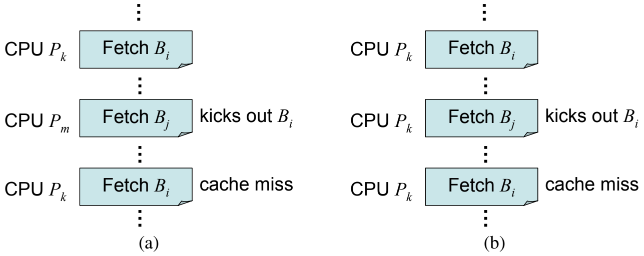

Figure 1: Examples of inter-CPU (a) and intra-CPU (b) harmful I/O prefetches. In (a), data block B_i brought into the memory cache (buffers) by CPU P_k is replaced by the prefetch of block B_j by CPU P_m, and B_i is referenced earlier than B_j. In (b), a CPU's (P_k) prefetch kicks out from the cache one of its own data.

Therefore, to be successful, an I/O prefetching scheme should be able to take these inter-thread data sharing patterns into account. In particular, if, in a given execution phase, a certain set of threads share significant amount of disk-resident data among them, one can use only a single I/O prefetcher for all of them, thereby reducing the impact of harmful prefetches.

This paper proposes an adaptive I/O prefetching scheme for disk-intensive parallel applications. The proposed scheme obtains inter-thread data sharing information using profiling and, based on the captured patterns, divides the threads into clusters and assigns a separate (customized) I/O prefetcher (thread) for each cluster. Since the data sharing patterns across threads change during execution, these clusterings and associated I/O prefetchers also change. That is, depending on the program phase in question, we may have a different number of I/O prefetchers assigned to different sets of threads. Therefore, this scheme is called "adaptive," as opposed to alternate prefetching schemes that fix the number of prefetchers for the entire duration of application execution.

Our approach brings benefits over conventional I/O prefetching where each CPU performs its own prefetches from disks, independently of the others. First, our approach reduces the additional cost of issuing prefetch instructions, as in our case for each set of threads a single prefetcher is reserved and thus other CPUs do not waste cycles in issuing prefetch calls. Second, and more importantly, in our approach for each shared data block a single prefetch is issued. Therefore, we can expect significant reductions in the number of harmful prefetches caused by multiple prefetches to the same data. Third, threads using shared data coordinate their accesses with each other, as they all must synchronize with the helper prefetch thread at the same point. This increases the chances that they will all find the shared data in the storage cache.

To test the effectiveness of our proposed I/O prefetching scheme, we implemented it using a compiler and the PVFS file system [25] running on top of Linux, and compared its performance against a number of alternate I/O prefetching schemes. Among the schemes tested are no prefetching case, a simple extension of compiler directed I/O prefetching to multiple CPUs (which we call "independent I/O prefetching" in this paper), assigning a fixed CPU for prefetching on behalf of all remaining CPUs (which do not perform prefetching), and assigning a small set of CPUs for prefetching. The empirical data collected clearly underline the importance of adapting I/O prefetching based on program phases. Specifically, our proposed scheme improves performance, on average, by 19.9%, 11.9% and 10.3% over the cases without I/O prefetching, with independent I/O prefetching, and with one CPU prefetching, respectively, when 8 CPUs are used.

The next section presents empirical evidence to motivate our approach. Section 3 summarizes the original compiler-directed I/O prefetching scheme proposed in [33]. Section 4 discusses the details of our adaptive I/O prefetching scheme. Sections 5 and 6 present experimental setup and collected results, respectively. Section 7 discusses related work, and Section 8 concludes the paper.

2. EMPIRICAL MOTIVATION

In this section, we present results from our experiments with four I/O-intensive applications to motivate the approach presented in this paper. The details of our experimental setup and applications will be given later. All applications have been parallelized using varying number of threads; each thread is assigned to a separate CPU, and all CPUs share the same storage cache (see Figure 6), which is 150MB (later we also present results with other cache sizes). Since we consider only one-to-one mappings between the CPUs and threads, in the remaining part of the paper, we use the terms "thread" and "CPU" interchangeably. Our first set of results are given in Figure 2 and plots the speedup curves under different CPU counts. These application codes are reasonably optimized for I/O (using source level techniques such as collective I/O [35]) but they do not use I/O prefetching. It can be observed from this plot that speedup of these applications is not scalable with the number of CPUs. As an example, when we use 16 CPUs, the speedups we achieve in applications HF, 3D-vis, Cholesky, and Mgrid are 9.3, 6.4, 10.4, and 6.5, respectively. In fact, the difference between the speedup numbers obtained using 14 CPUs and 16 CPUs is negligible. While these speedup results are collected with these four applications, poor scalability of I/O-intensive applications is a well-known fact. For example, the work in [22] reveals that lack of I/O scalability severely limits potential speedups that could be achieved in I/O-intensive applications. As far as our research is concerned, the main take away message from these results is that a couple of CPUs can be taken away from the application without affecting its speedup too much. Clearly, as we move to larger number of CPUs (e.g., 128-256 range), one can expect that taking away even 7 or 8 CPUs from an application would not affect its performance too much.

Our second set of results focus on the performance of I/O prefetching in parallel applications. Figure 3(a) presents the percentage improvements in total execution cycles of our four applications due to I/O prefetching. Specifically, each bar corresponds to the performance improvement brought by the I/O prefetching scheme in [33] over the no-prefetch case. In the prefetching case, we applied the scheme in [33] to each thread of the application independently (a summary of the I/O prefetching scheme in [33] is given later in Section 3). An important observation from these results is that the effectiveness of I/O prefetching diminishes dramatically as the number of CPUs to execute the application code increases. For example, with application HF, the improvement brought by prefetching is about 29.5% when a single CPU is used, whereas the same is only 1.3% with 12 CPUs. In fact, I/O prefetching degrades the overall performance (as compared to the no-prefetch case) in all four applications when 15 or 16 CPUs are used. To understand why this happens, we collected additional statistics capturing the prefetch-related interactions among the CPUs[1]. Figure 3(b) gives the fraction of harmful I/O prefetches. As stated earlier, we define a "harmful prefetch" as a prefetch that leads to the removal of a data block from the memory cache and the prefetched data block is referenced only after the reference to the removed block. We see from Figure 3(b) that, the contribution of harmful prefetches increases with the increasing number of CPUs. This in a sense can be expected, as more CPUs are used for executing the application, higher the chances that CPUs will replace each other's data from the shared storage cache when they prefetch. We need to mention however that harmful prefetches alone may not be the only reason for the sharp degradation in performance as we increase the number of CPUs. For example, we also noticed during our experiments that the negative interactions even among normal disk fetches to the memory cache also tend to increase with the large number of

[1] Since the shared storage cache we implemented is managed by software, we modified it to collect harmful prefetches. Specifically, we increment the counter when a prefetch leading to the removal of a data block from the cache and the prefetched block is referenced only after the reference to the removed block.

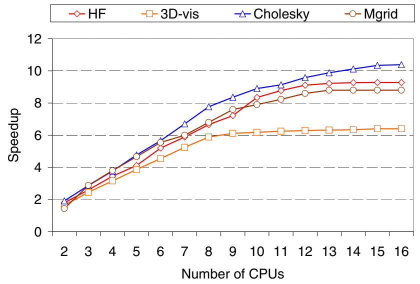

Figure 2: Speedup numbers when all CPUs sharing the same I/O node are used.

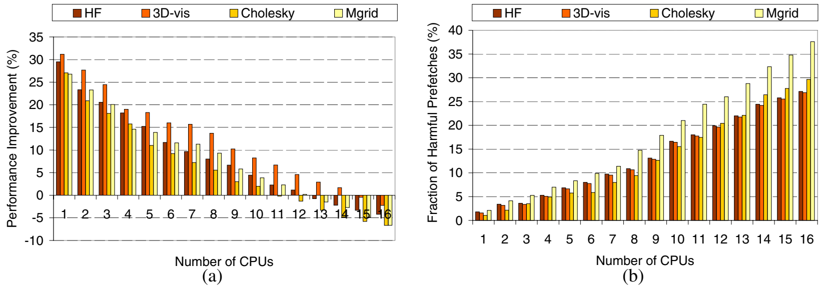

Figure 3: (a) Percentage improvements brought by I/O prefetching. (b) Percentage of harmful prefetchings. All savings are with respect to the case without I/O prefetching.

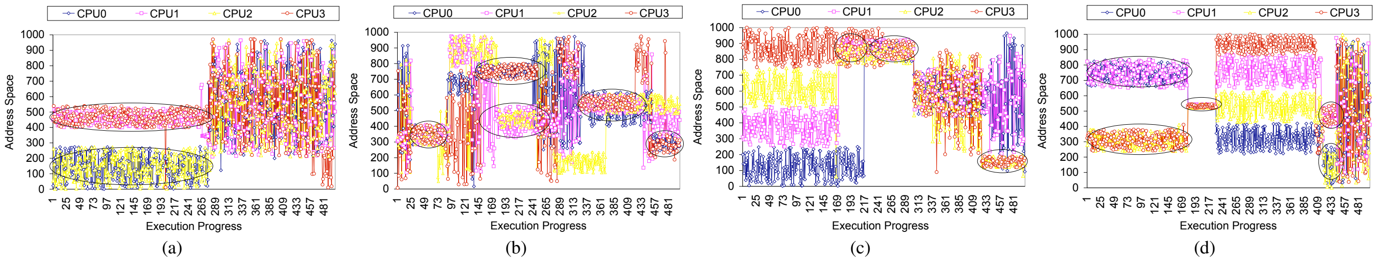

Figure 4: Data access patterns of four I/O-intensive applications obtained through profiling. (a) HF (b) 3D-vis (c) Cholesky (d) Mgrid. In 3D-vis, the pattern shown repeats itself multiple times. In HF, only the most time-consuming portion of the code is shown. The ovals are used to capture sample patterns for which we can use a common prefetcher (one per oval).

CPUs. Nevertheless, the results presented in Figures 3(a) and 3(b) illustrate a strong correlation between the performance degradation and the fraction of harmful I/O prefetches.

Our third set of results study the I/O access patterns of our applications focusing on shared data. Figure 4(a) illustrates an interesting scenario when an application (HF) is executed using four CPUs. The x-axis of this figure denotes the execution progress and the y-axis captures the addresses of the data elements. In obtaining this graph, the total application execution period is divided into 500 epochs, and the addresses of the accessed data elements are recorded. Note that the address space shown in Figure 4 is rather a file offset since all the array data is stored on the disk. Therefore, Figure 4 shows the access clustering patterns in terms of file domain instead of memory or cache address space. We can identify two distinct execution phases in this graph, which correspond to two different functions in the application code that consume nearly 95% of the total application execution time. In the first execution phase (function), CPU0 and CPU2 share the same small set of data elements and similarly CPU1 and CPU3 access a lot of common data elements, which constitute a small subset of the total address space. In comparison, in the second phase which starts around epoch 270, a much larger set of data are accessed (note that some of these data are accessed by more than one CPU; however, the total data range is too large). We believe that an I/O prefetching strategy can be tuned by exploiting this execution profile. Specifically, for the first phase of this application, it may be a good idea to use an I/O prefetcher (thread) for CPU0 and CPU2 and another I/O prefetcher for CPU1 and CPU3. In this case, the application threads running on CPU0, CPU1, CPU2 and CPU3 do not perform any I/O prefetching (the prefetch threads perform it on behalf of them). Note that, in this phase, since CPU0 and CPU2 (and similarly CPU1 and CPU3) share a lot of data between them, allocating a common prefetcher will cut the number of prefetches and reduce the chances for harmful prefetches. On the other hand, in the second phase, it may be more beneficial to employ a separate I/O prefetcher for each of the CPUs (each I/O prefetcher in this phase can be integrated with its associated application thread, as in [33]). Figure 4(b) shows the execution profile of another I/O-intensive application (3D-vis). In this application, we observe even more phases with interesting data sharing patterns. For instance, between epochs 25 and 90 (which corresponds to a large loop nest in the application), all four CPUs access a small set of data and can potentially share the same I/O prefetcher. A similar behavior can also be observed between epochs 320 and 430. In these portions, it may be a good idea to employ only a single I/O prefetcher that prefetches data on behalf of all four CPUs. On the other hand, between epochs 450 and 500, it may be a good idea to have a single I/O prefetcher devoted to CPU0 and CPU3; the remaining CPUs can have their own private prefetchers. The graphs in Figures 4(c) and 4(d) present the execution profiles for our remaining two applications and one can make similar observations from these plots as well. Although not presented here, we also observed similar clustering patterns when larger number of CPUs are used.

Considering the results presented in Figures 2 through 4, we can reach the following conclusions. The inter-CPU data sharing pattern for a given I/O-intensive application varies significantly during the course of execution. Given the poor performance of independent I/O prefetching in large CPU counts (wherein each CPU issues its I/O prefetches independently), it is clear that we have to take inter-CPU data sharing patterns into account to achieve acceptable program performance through I/O prefetching. Instead of allowing each CPU to perform I/O prefetching independently (i.e., execute I/O prefetcher threads in addition to application threads), one option is to reserve a couple of CPUs to do prefetching on behalf of the others which execute application code without issuing any prefetch call. Based on our results above (Figures 2 and 3), we know that this is unlikely to hurt scalability of the parallel application. In the rest of this paper, we present and experimentally evaluate such an adaptive I/O prefetching scheme which modulates the number of the threads to use for I/O prefetching based on the inter-thread data sharing patterns.

3. COMPILER-DIRECTED I/O PREFETCHING

While there exist several I/O prefetching algorithms published in literature [27, 2, 33, 36, 19, 11, 15], the one used in this work is inspired by the work done by Mowry et al [33]. The original algo-

```
int $X[0..N-1]$;
int $Y[0..N-1]$;
int $Z[0..N-1]$;
for $i = 0$ to $N-1$ {
  $Z[i] = X[i] \times Y[i]$;
}
```

(a)

```
int $X[0..N-1]$, $Y[0..N-1]$, $Z[0..N-1]$;
prefetch($X$, 0, $P$);    prefetch($Y$, 0, $P$);
for $t = 0$ to $\lfloor N/P \rfloor - 1$ {
  prefetch($X$, $(t+1) \times P$, $P$);
  prefetch($Y$, $(t+1) \times P$, $P$);
  for $i = 0$ to $P-1$, 1
    $Z[t \times P + i] = X[t \times P + i]$
        $\times Y[t \times P + i]$;
}
for $j = \lfloor N/P \rfloor \times P$ to $N-1$
  $Z[j] = X[j] \times Y[j]$;
```

(b)

Figure 5: An example that illustrates compiler-directed I/O prefetching. (a) Original code fragment. (b) Compiler-generated code with explicit I/O prefetch calls inserted. The syntax of the I/O prefetch call is similar to that of a regular I/O call, i.e., `prefetch (array_name, offset, size)`**, where the second parameter indicates the location and the third one captures the length of the data.**

rithm has actually been proposed for improving hardware cache behavior for memory-resident data sets [32], and has later been extended to implement I/O prefetching targeting virtual memory based execution environments. We adapted this algorithm to work with explicit disk I/O.

Figure 5 illustrates an example of this compiler-directed I/O prefetching scheme. For the sake of clarity, we omit the actual file I/O statements (the PVFS [25] calls in our case). In this example, three N-element "disk-resident" arrays (X, Y, and Z) are accessed using three references ($X[i]$, $Y[i]$, and $Z[i]$). P denotes the data block size, which is assumed to be the unit for I/O prefetching (i.e., an I/O prefetch targets a data block of size P). Figure 5(a) shows the original loop (without I/O prefetching), and Figure 5(b) illustrates the compiler-generated code with prefetch calls embedded. Note that, in order to perform prefetches with the specified block size (P), the loop is modified to operate on a block size granularity. As can be seen in the compiler generated code of Figure 5(b), the outermost loop iterates over individual data blocks, whereas the innermost loop iterates over the elements within a block (this particular transformation is called strip-mining [38]). This way, it is possible to prefetch a data block and operate on the data elements it contains. The first set of prefetch statements in the compiler-generated code is used to load the first set of data blocks into the memory cache prior to the main loop execution. In the steady state, within the loop, we first issue the 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.

We now briefly discuss the compiler analysis required for implementing this I/O prefetching. First, the compiler analyzes the given application code and predicts the future data access patterns. This is done using "data reuse analysis", a technique developed originally for conventional cache locality optimization [37]. This analysis identifies how a given data element is accessed by different iterations and statements of a loop nest, and captures the reuse distances (in terms of loop iterations). In [33], misses are isolated through loop-splitting and prefetches are scheduled using software pipelining based on the data locality information generated by the compiler. In their I/O prefetching algorithm, one of the key modifications to the original algorithm (which targets memory resident data sets) is to limit the prefetches only across the outermost loop nest. This follows from the fact that cache lines have relatively small sizes when compared to a page (unit of prefetch in the I/O prefetching algorithm of [33]), hence inner loop nests often access less data than a page (in our case, block) size. In deciding the loop splitting point, the prefetching algorithm in [33] takes into account the estimated I/O latencies as well.

In our implementation of this I/O prefetching algorithm, we have a layer in the file system that monitors the prefetch requests and filters unnecessary prefetches as much as possible (a similar runtime layer is also used in [33]). In this layer, a "bitmap" is maintained to capture the set of data blocks that are already in the memory cache. Whenever a prefetch is to be issued to the disk, the corresponding bit is checked to see whether the block in question is already in the memory cache, and if this is the case, that prefetch is suppressed. In this way, a significant number of useless I/O prefetches can be eliminated. We want to emphasize that, while our experiments use this particular I/O prefetching algorithm, its choice is really orthogonal to the main focus of this paper. In other words, as far as its applicability is concerned, our approach can be used along with any existing I/O prefetching algorithm. Clearly, the savings achieved by our schemes will be dependent on the underlying prefetching algorithm used, and in fact, we believe our approach can bring larger benefits when it is used along with simpler I/O prefetching algorithms (instead of a compiler-directed one). The reason for this is that the algorithm in [33] inserts prefetches very carefully taking into account loop-specific I/O behavior and estimated I/O latencies. As a result, it inserts few useless prefetches and most of such useless prefetches are filtered before they are actually issued to the disks. Simpler schemes on the other hand would tend to insert more useless prefetches (some of which will also be harmful prefetches).

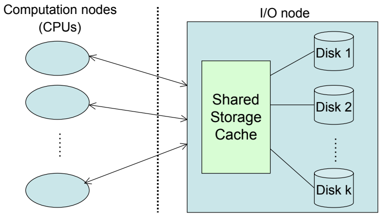

Figure 6: I/O system abstraction.

4. OUR APPROACH

4.1 I/O System Abstraction

Figure 6 shows the architecture of a typical modern storage system interfaced with a computation system. The computation nodes are connected to the disk storage system through the shared storage cache. The purpose of the storage cache is to store a subset of the disk-resident data. If the requested data is found in the storage cache, then the access time is much less than the case when the disk needs to be accessed. Therefore, proper maintenance of this shared cache is extremely important.

4.2 High Level View

Figure 7 gives a high-level view of our approach which consists of three steps (components). In the first step, we profile the code and identify the data sharing patterns among the different CPUs. The outcome of this step can be shown in the form of plots, as in Figures 4(a) through 4(d), which help us identify the number and types of the I/O prefetchers to use. In the second step, we associate the identified sharing patterns to code sections. Note that, while the first step gives us the thread groupings (i.e., which set of threads should be assigned a common I/O prefetching thread), the second step tells us the program segments where these groupings should be considered. As a concrete example, let us consider once more the execution profile shown in Figure 4(a) where we can easily identify two phases, which correspond to two different functions called by the main(.) routine of this application. Each of these functions has a very large loop nest in its body. For the first phase (loop nest), since CPU0 and CPU2 share considerable amount of data, we assign a common prefetcher for CPU0 and CPU2 and for similar reasons, we assign a common prefetcher for CPU1 and CPU3.

Note that we need a metric using which we can decide whether two (or more) CPUs can work with a common I/O prefetcher in a given phase. The metric we use for this purpose is called the

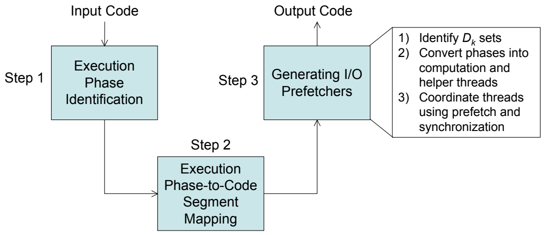
Input Code
Output Code
Step 1
Execution Phase Identification
Step 3
Generating I/O Prefetchers
1) Identify D_k sets
2) Convert phases into computation and helper threads
3) Coordinate threads using prefetch and synchronization
Step 2
Execution Phase-to-Code Segment Mapping

Figure 7: High-level view of our approach.

"sharing density", and gives the ratio between the number of data elements shared by CPUs and the total number of distinct data elements accessed in that phase. If this number exceeds a preset threshold value, those CPUs are given a common I/O prefetcher. At the end of this assignment, a CPU that is not assigned a common prefetcher performs its own I/O prefetching (similar to [33]). For example, if the sharing density threshold is set to 80% (the default value used in our experiments), two CPUs that have a sharing density of 80% or higher in a phase are assigned the same I/O prefetcher in that phase.[2] While in this paper we employ a profile-based approach to capture inter-CPU data sharing patterns and determine the program segments for which prefetchers are embeded, one can also employ, where possible, static program analysis to capture data sharing patterns across CPUs. The remainder of our approach (that is, the third component in Figure 7), which generates I/O prefetchers, is actually independent of how data sharing patterns have been identified.

4.3 Generating I/O Prefetchers

In this section, we explain how the I/O prefetchers are obtained for a given cluster of CPUs in a code segment (phase) identified by the previous steps. What we mean by "cluster" is a set of CPUs such that, after our approach, one of them prefetches data on behalf of the others. We want to remind the reader that we assume one thread per CPU. Throughout our discussion, we assume that an identified program phase has only single loop nest (which can contain multiple loops). If a phase contains more than one nest, we apply our approach to each of them.

The main objective of our compiler algorithm is to transform each identified phase from the earlier step into "computation threads" and "helper threads". In our approach, the computation threads perform only computation, whereas the helper threads perform all I/O prefetches on behalf of the computation threads. This is accomplished by three steps which are also shown in Figure 7. The details of the third step are discussed below.

4.3.1 Identifying Data Elements to be Prefetched

Let us focus on a phase (the corresponding code segment) and a CPU cluster of size B. As stated earlier, a cluster of B CPUs means that one CPU will perform prefetches from the disks, i.e, run the helper thread, and the remaining $B-1$ CPUs will perform computations, i.e., execute computation threads. Let us assume for now $B \geq 2$. We start by dividing the phase into m *slices* and use $\mathcal{I}_{i,k}$ to denote the set of iterations assigned to computation thread i in slice k, where $1 \leq i \leq B-1$ and $1 \leq k \leq m$. We can compute $\mathcal{D}_{i,k}$, the set of data elements that will be accessed by computation thread i in slice k as follows:

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

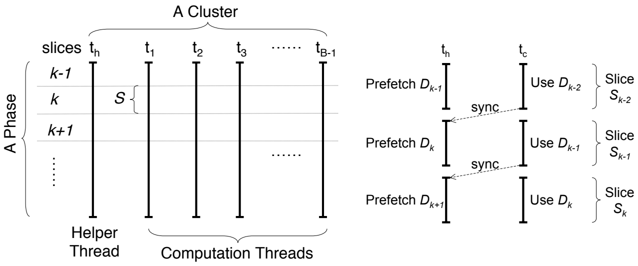

Figure 8: (a) Overview of how helper thread and computation threads are scheduled from a single cluster perspective. Note that a phase identified in the profiling step contains multiple slices. (b) Synchronization mechanism between helper thread (t_h) and computation thread (t_c).

In this formulation, $\mathcal{R}_{i,k}$ is the set of references to the disk-resident array; 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). Note that, since $\mathcal{I}_{i,k}$s within a particular slice are likely to share data elements, we can expect that:

$$\mathcal{D}_{x,k} \cap \mathcal{D}_{y,k} \neq \emptyset, \ \text{ for } \ x \neq y.$$

After obtaining $\mathcal{D}_{i,k}$ for each thread i in the k^{th} slice, we next determine the entire set of data elements accessed by the k^{th} slice, denoted as $\mathcal{D}_k$, by taking the union of $\mathcal{D}_{i,k}$ sets:

$$\mathcal{D}_k = \bigcup_{1 \leq i \leq B-1} \mathcal{D}_{i,k},$$

where B is the number of CPUs as stated earlier. Note that, depending on the degree of data sharing among threads, the size of $\mathcal{D}_k$ can be much smaller than the sum of the sizes of individual $\mathcal{D}_{i,k}$s, i.e., $|\mathcal{D}_k| \ll \sum_{i=1}^{B-1}\{|\mathcal{D}_{i,k}|\}$. In fact, higher the sharing density (as defined earlier), larger the difference between $|\mathcal{D}_k|$ and $\sum_{i=1}^{B-1}\{|\mathcal{D}_{i,k}|\}$. Note that we can build a $\mathcal{D}_k$ set for each disk resident array to which I/O prefetching will be applied in slice k. For each of these arrays, we insert separate prefetch instructions in the code.

4.3.2 Generating Codes for the Computation Threads and the Helper Thread

To generate code for inserting prefetch instructions, we need to make the slice boundaries explicit in the thread codes. In our implementation, this is achieved for the computation threads using strip-mining [38]. The work to be done for the helper thread is more involved. We first generate the addresses to be prefetched for the elements in each $\mathcal{D}_k$, and then insert prefetch instructions for these addresses. Clearly, we do not want to issue multiple prefetches for the same data block. In our implementation, we use the Omega Library [26] 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 $\mathcal{D}_k$. After this, these individual loops for different slices are combined to generate a compact code where the outer loop iterates over the different slices (k) 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, and the results were very satisfactory for the application codes we targeted. At this point, we have the codes for both computation threads and helper thread. The code for the helper thread also contains I/O prefetch calls, and both computation thread and helper thread codes are such that the slice boundaries are explicit to enable synchronization between the computation and helper threads, which is discussed next.

[2] A more accurate metric would take into account the length of the phase as well, since working with small phases can lead to code bloat. However, in the codes we used in this study, the phases correspond to different functions that contain multiple nested loops, and thus, they are very large. Therefore, using "sharing density" works fine. In our implementation, if a phase corresponds to a function that contains multiple, separate loop nests, we applied our prefetching strategy to each of them independently.

```
Input:
    $\mathcal{P}$ – an input program, $\mathcal{P} = (\mathcal{L}_1, \mathcal{L}_2, \cdots, \mathcal{L}_x)$,
        where $x$ is the number of phases in $\mathcal{P}$;
Output:
    $\mathcal{P}'$ – transformed program, $\mathcal{P}' = (\mathcal{L}'_1, \mathcal{L}'_2, \cdots, \mathcal{L}'_m)$;

$C_k$ – CPU cluster that exhibits accesses on shared data;
$B$ – the size of $C_k$;
$\mathcal{C}$ – all CPU clusters that belong to $y^{\text{th}}$ phase;
$T$ – minimum cluster size, default is 2;
$S$ – number of iterations used for strip-mining the original loop nest;

procedure gen_helper() {
  for each $C_k$, $C_k \in \mathcal{C}$ {
    if ($B$ of $\mathcal{C}_k \leq T$)
      for each computation thread $t_i$, $t_i \in \mathcal{C}$
        apply conventional I/O prefetch scheme such as [33];
    else { /* This is the case we want to tackle */
      clone the original computation thread and tag it as helper thread;
      assign $B - 1$ CPUs to the computation threads;
      assign 1 CPU to the duplicated helper thread;
      compute new lower and upper bound for the computation threads;
      strip-mining both main and helper thread using $S$;
      for each thread $t_i$, $t_i \in \mathcal{C}$ { compute $\mathcal{D}_{i,k}$ };
      call Omega_library to enumerate iterations of each $\mathcal{D}_{i,k}$;
      for each array $\in \mathcal{L}_j$ { compute $\mathcal{D}_k = \bigcup_{1 \leq i \leq B-1} \mathcal{D}_{i,k}$; };
      for each array $\in \mathcal{L}_j$ {
        /* $|\mathcal{D}_k|$ is the size of array data to be prefetched */
        emit "prefetch();" for $\mathcal{D}_k$ in the helper thread;
      }
      emit "synch()" for both computation and helper threads;
    }
  }
}
main() {
  for each phase $\mathcal{L}_k$, $\mathcal{L}_k \in \mathcal{P}$ {
    let $\mathcal{C}$ be the CPU cluster in $\mathcal{L}_k$;
    read profiler information for $\mathcal{C}$;
    call gen_helper();
  }
}
```

Figure 9: Compiler algorithm for transforming a given code to insert I/O prefetch instructions.

4.3.3 *Synchronizing the Computation Threads and the Helper Thread*

Figure 8 illustrates the interaction in a cluster among the computation threads and the helper thread which prefetches on behalf of those computation threads. As explained above, the phase in question is divided into slices and each slice contains S iterations, as shown in Figure 8(a). The synchronizations occur across slice boundaries. The goal is to ensure that when the computation threads start operating on slice k, all the prefetches (that bring data in $\mathcal{D}_k$ to the shared storage cache) are completed. As shown in Figure 8(b), this is achieved in our approach by inserting synchronization calls among the helper and computation threads. More specifically, at the beginning of slice $(k-2)$, the helper thread issues the prefetch calls for data in $\mathcal{D}_{k-1}$. However, before it issues the prefetch calls for data in $\mathcal{D}_k$, it synchronizes with the computation threads indicating that all the computation threads are done with their computations in slice $(k-2)$ and are ready to proceed to slice $(k-1)$. Once the synchronizations take place, the helper thread starts prefetching the data in D_k and the computation threads start operating on the data in D_{k-1} (see Figure 8(b)).

Figure 9 gives our compiler algorithm explained so far in a pseudo code form. The main() procedure takes an input program, $\mathcal{P}$, along with the profile information ($\mathcal{C}$ for each loop nest) and the number of CPUs (B) in each identified cluster, and outputs the transformed version of the program that consists of the helper thread and computation threads.

4.3.4 *Discussion*

It is important to note that we target array-intensive applications, and in these codes, the data access/sharing patterns do not change much with input data. Therefore, we can expect that profiling works well with these codes, and in fact, in our experiments, the input data sets using for actual execution were different from those used in profiling. Also, we want to mention that prefetching technique itself is not useful at all for the applications that do not show any regularity of accesses, e.g., random access. Many I/O-intensive applications spent quite amount of time in I/O and they show regular access pattern, which makes sense for employing our helper thread based I/O prefetching for reducing harmful prefetches. Although there is certain amount of temporal locality in the data brought to the memory and current computers have unprecedented memory capacity, we believe that applications based on out-of-core kernels are still needed to solve even larger applications. We also want to mention that our technique can be applied to other type of applications such as memory-intensive codes that access the shared L2 cache in CMP.

Our approach, as explained so far, generates a helper thread for each cluster. As a result, for each cluster, we lose a CPU, which can hurt performance for small sized clusters. We explored two approaches to address this issue. The first approach is to run the helper thread of a cluster in one of the CPUs of that cluster. This means that one CPU in the cluster will execute both its share of the application code (a computation thread) and the prefetching code (for all CPUs). Our experiments with this approach did not generate good results. In fact, the results obtained with this version were not as good as those obtained through independent I/O prefetching. The second approach is to go back to independent prefetching if the cluster size is lower than a preset threshold value. For example, we found that when the cluster size is two, it is better to have each CPU to prefetch its own data (rather than running the application code in one CPU while the other one performs I/O prefetching). On the other hand, when the cluster size is three, our approach, which uses two CPUs for computation threads and reserves the last one for prefetching, generated better results. This was also the case when the cluster size is larger than three. Therefore, we set the minimum cluster size for our approach to be applied to three in our experiments.

In our approach, we used profiling to detect the CPU/thread clustering that accesses the shared data. And this information may not be available during static compilation time because many scientific kernels (mostly loop nests) are written such that they are parallelized according to the number of processors/CPUs given as an input. The amount of profile data is also limited because we only collect I/O request to disk-resident data set, not the every addresses accessed by each thread. For less regular codes that do not have easily analyzed or transformed loop nests, we still believe that our approach can be applicable to some extent as long as a generated helper thread is able to interact with runtime system, which collects the information on what to prefetch and which CPUs access the shared data.

Lastly, as our approach reduces both the number of harmful prefetch instructions and the amount of duplicate data blocks brought in the cache, we expect that it also incurs less paging in the underlying operating system.

4.4 Example

We consider the example code fragment in Figure 10, which contains three separate loop nests. For the illustrative purposes, let us assume that there are 16 CPUs and each of these nests is parallelized over these CPUs. For the sake of clarity, we omit the actual file I/O (PVFS) statements. All arrays (X, Y, Z, A, R, and M) are assumed to be disk-resident. The first loop nest contains a computation that references Z, X and Y using three references ($X[i, j]$, $Y[i, j]$, and $Z[i, j]$), and similarly the second and third loop nests contain computations that refer to Z, A, R and M. Based on the information from our profiling step, which indicates the data sharing pattern, we can identify three distinct phases in this code fragment, each corresponding to one of the loop nests. The first loop nest has accesses to the distinct elements of the arrays in each iteration and

```
for i=0 to 63 {       /* 1st loop nest */
   for j=0 to N − 1
      Z[i, j] = X[i, j] × Y[i, j];
}
for i=0 to 63 {       /* 2nd loop nest */
   for j=0 to N − 1 {
      k = (int) i / 32;
      Z[i, j] += A[i, j] × M[k, j];
   }
}
```

```
for i=0 to 63 {       /* 3rd loop nest */
   for j=0 to N − 1 {
      k = (int) log2((int)i/4) ;
      Z[i, j] += R[k, j];
   }
}
```

Figure 10: Original code fragment with three loop nests.

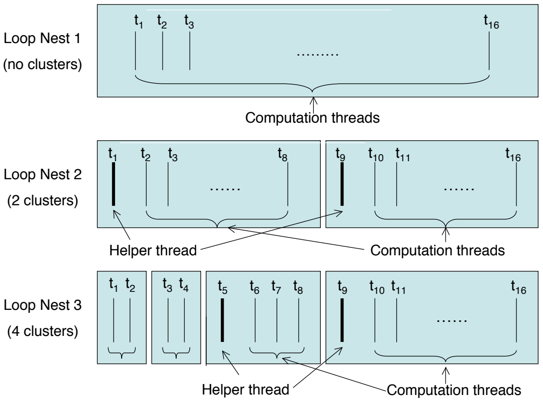

Figure 11: Computation and helper thread assignments in different loop nests.

hence there is no data sharing among the data elements accessed by different threads. In contrast, in the second loop nest, the first half of the outer loop (i loop index) iterations access some common data ($M[k, j]$), and the second half of the iterations also share similar data among themselves. As a result, two clusters of data sharing (and thus two CPU clusters) can be clearly identified. The third (last) loop nest also exhibits similar sharing but the corresponding clusterings are quite different from those in the second loop nest. The clustering according to the outer loop iterations is as follows: 12.5%, 12.5%, 25%, 50%, which means that the first 12.5% iterations share the same data and so do the next 12.5%, the next 25% and then the last 50%. We have chosen this particular example with these data access patterns for the purpose of clearly illustrating and conveying the concept of clustered data sharing among threads. The key point we wish to make here is the change in the clustering pattern as the program execution goes through the different phases (loop nests).

Figure 11 gives a pictorial view, under our approach, of the thread distribution structure in the three loop nests of the program. When the first loop nest is in execution (there is no data sharing and hence no clustering), all threads (t_1 to t_{16}) are computation threads doing their own I/O prefetching (similar to [33]). As the execution proceeds to the second loop nest, since there are two identifiable clusters, one helper thread each (t_1 for the first cluster, t_9 for the second cluster) is assigned to the clusters and are involved in doing the prefetching for the whole cluster. Finally, in the third loop nest, threads t_1 through t_4 perform their own I/O prefetches because the first two clusters have two CPUs each. The remaining two clusters follow our adaptive prefetching scheme and get assigned 4 and 8 CPUs, respectively, with 1 helper thread per each cluster.

Figure 12(a) illustrates the traditional compiler-directed I/O prefetch case used for CPU1 in the first loop nest. In the first loop, there is no sharing among CPUs, and as a result, we apply the traditional prefetching scheme to the code fragment assigned to each CPU. In order to perform prefetches with the specified block size (P), the loop is modified to operate on a block size granularity. The outermost loop iterates over individual data blocks, whereas the innermost loop iterates over the elements within a block. The code fragments for the remaining 15 CPUs have similar structures.

For the second loop nest, our algorithm, after identifying the clustering pattern, assigns a helper thread to each of the two CPU clusters. Since this takes away 2 CPUs (recall that we assign one thread per CPU), the iterations are redistributed (parallelism is retuned) among the remaining 7 threads in each cluster (see Figures 12(b) and (c)). The third loop nest in this example code fragment has a more complex clustering pattern. We use the traditional I/O prefetch insertion for the first two clusters since they consist of only 2 threads and taking away one of the them for prefetching purposes would adversely affect the performance (based on our discussion in Section 4.3.4). One of the threads belongs to the first cluster of the third loop nest is given in Figure 12(d). The remaining two (third and fourth) clusters are assigned one helper thread each and the iterations are redistributed among the remaining threads. Figure 12(e) illustrates the structure of the helper thread, and Figure 12(f) shows the computation thread in the same cluster. This thread is intended to perform only the computation since it has a helper thread that performs prefetching for it. Similarly, Figures 12(g) and (h) show the helper and computation thread for the second cluster in the third loop nest. When we look at the helper threads for the second (Figure 12(b)) and the third (Figure 12(e)) loop nests, an important difference can be noticed. The helper thread for the second loop nest has a single prefetch instruction for the shared data and a loop of prefetch instructions to prefetch unshared data while the helper thread for the third loop nest has only one prefetch instruction since the clusters do not access unshared data.

5. EXPERIMENTAL SETUP

We used four I/O-intensive applications in this study:

• HF: The Hartree-Fock (HF) method is an approximate method for the determination of the ground-state wave function and ground-state energy of a quantum many-body system. At the heart of the method is the construction of the Fock matrix using an iterative procedure. At each iteration, the Fock matrix is updated using integral calculations. The results of these integrals in the current iteration are stored on disk and read by the next iteration. The molecule sizes used in our setting resulted in a total dataset size of 12.4GB. Our implementation of this code closely follows that of [22].

• 3D-vis: This is a visualization code for 3D image data such as CT and MR. The code includes generation of 3D surface models and 3D tetrahedral models, computation of iso-surfaces, and direct volume rendering. The datasets manipulated by the code are disk-resident and the current implementation we have includes additional optimizations such as collective I/O [35] to maximize the I/O performance as much as possible. The dataset sizes used in our experiments varied between 11.1GB and 16.8GB.

• Cholesky: This application implements the factorization and solution of a dense system that stores its matrices on disks. Our implementation closely follows the one discussed in [3] and the subportions of the main disk resident matrix are transferred to memory as needed. As in the case of 3D-vis, the I/O behavior of the application has been carefully optimized as much as possible using known techniques such as collective I/O [35]. The total size of the data manipulated by this benchmark is about 11.7GB.

• Mgrid: This is the out-of-core version of an application that appears in both [39] and [16]. This application demonstrates the capabilities of a simple multigrid solver in computing a three dimensional potential field. In this application, in addition to echoing some of the inputs, the main part of the output gives the smoothed approximate inverse. As in the case of Cholesky and 3D-vis, collective I/O is used for maximizing disk performance. In a typical run, the total data size manipulated by this application is about 13.4GB.

We made our experiments using PVFS, the Parallel Virtual File System [25], which runs on top of a Linux cluster. PVFS is mainly a user-level implementation, i.e., there is a library (libpvfs) linked

```
$P_{id}$ = 1;     $B$ = 16;
$lb = (P_{id}$-1) $\times$ (64/$B$); /* lower loop bound */
$ub = (P_{id} \times$ (64/$B$))-1; /* upper loop bound */
for $i = lb$ to $ub$ {
  prefetch($X$, i, $P$);
  prefetch($Y$, i, $P$);
  for $t$=0 to $\lfloor N/P \rfloor - 1$ {
    prefetch($X$, $(t+1) \times P$, $P$);
    prefetch($Y$, $(t+1) \times P$, $P$);
    for $j$=0 to $P$–1
      $Z[i, t \times P + j]$ =
        $X[i, t \times P + j] \times Y[i, t \times P + j]$;
  }
  for $j$=$\lfloor N/P \rfloor \times P$ to $N$–1
    $Z[i, j] = Z[i, j] \times Y[i, j]$;
}
```

(a) Loop Nest 1, CPU1

```
$N_{itr}$ = number of iterations assigned to this cluster;
$lb$ = first iteration of this cluster;
$ub = lb + N_{itr}$;   $B$ = 8;   $BB = N_{itr}/B$;
for $i$= $lb$ to $ub$ {
  /*prefetch the shared reference(const=(int)$i$/32) only once*/
  prefetch($M$, const, $P$);
  /*then prefetch the unshared data for all cores*/
  for $x$ = 0 to $B - 1$
    prefetch($A$, $BB \times x + i$, $P$);
  for $t$=0 to $\lfloor N/P \rfloor - 1$ {
    prefetch($M$, ($t$+1)$\times P$, $P$);
    for $x$=0 to $B - 1$ {
      prefetch($A$, $BB \times x + i$, $P$);
      synch(syncvar1);
    }
  }
}
```

(b) Loop Nest 2, CPU1 (helper thread)

```
$lb$ = first iteration assigned to CPU2;
$ub$ = last iteration assigned to CPU2;
for $i$= $lb$ to $ub$ {
  $k$ = (int) $i$ / 32;
  for $t$=0 to $\lfloor N/P \rfloor - 1$ {
    for $j$=0 to $P$–1
      $Z[i, t \times P + j]$ +=
        $A[i, t \times P + j] \times$
        $M[k, t \times P + j]$;
    synch(syncvar1)
  }
  for $j$=$\lfloor N/P \rfloor \times P$ to $N$–1
    $Z[i, j]$ += $A[i, j] \times M[k, j]$;
}
```

(c) Loop Nest 2, CPU2

```
$P_{id}$ = 1;     $B$ = 16;
$lb = (P_{id}$-1) $\times$ (64/$B$); /* lower loop bound */
$ub = (P_{id} \times$ (64/$B$))-1; /* upper loop bound */
for $i = lb$ to $ub$ {
  $k$ = (int) $log_2$((int)$i$/4) ;
  prefetch($R$, $k$, $P$);
  for $t$=0 to $\lfloor N/P \rfloor - 1$ {
    prefetch($R$, $(t+1) \times P$, $P$);
    for $j$=0 to $P$–1
      $Z[i, t \times P + j]$ += $R[k, t \times P + j]$;
  }
  for $j$=$\lfloor N/P \rfloor \times P$ to $N$–1
    $Z[i, j]$ += $R[k, j]$;
}
```

(d) Loop Nest 3, CPU1

```
$N_{itr}$ = number of iterations assigned to this cluster;
$lb$ = first iteration of this cluster;
$ub = lb + N_{itr}$;
for $i$= $lb$ to $ub$ {
  /* prefetch the shared reference
       (const=(int)log2($i$/4)) only once */
  prefetch($R$, const, $P$);
  for $t$=0 to $\lfloor N/P \rfloor - 1$ {
    prefetch($R$, $(t+1$, 1);
    synch(syncvar1);
  }
}
```

(e) Loop Nest 3, CPU5 (helper thread)

```
$lb$ = first iteration assigned to CPU6;
$ub$ = last iteration assigned to CPU6;
for $i = lb$ to $ub$ {
  $k$ = (int) $log_2$((int)$i$/4) ;
  for $t$=0 to $\lfloor N/P \rfloor - 1$ {
    for $j$=0 to $P$–1
      $Z[i, t \times P + j]$ += $R[k, t \times P + j]$;
    synch(syncvar1);
  }
  for $j$=$\lfloor N/P \rfloor \times P$ to $N$–1
    $Z[i, j]$ += $R[k, j]$;
}
```

(f) Loop Nest 3, CPU6

```
$N_{itr}$ = number of iterations assigned to this cluster;
$lb$ = first iteration of this cluster;
$ub = lb + N_{itr}$;
for $i$= $lb$ to $ub$ {
  /* prefetch the shared reference
       (const=(int)log2(i/4)) only once */
  prefetch($R$, const, $P$);
  for $t$=0 to $\lfloor N/P \rfloor - 1$ {
    prefetch($R$, ($t$+1) $\times P$, $P$);
    synch(syncvar2);
  }
}
```

(g) Loop Nest 3, CPU9 (helper thread)

```
$lb$ = first iteration assigned to CPU10;
$ub$ = last iteration assigned to CPU10;
for $i$= $lb$ to $ub$ {
  $k$ = (int) $log_2$((int)$i$/4);
  for $t$=0 to $\lfloor N/P \rfloor - 1$ {
    for $j$=0 to $P$–1
      $Z[i, t \times P + j]$ += $R[k, t \times P + j]$;
    synch(syncvar2);
  }
  for $j$=$\lfloor N/P \rfloor \times P$ to $N$–1
    $Z[i, j]$ += $R[k, j]$;
}
```

(h) Loop Nest 3, CPU10

Figure 12: Example application.

to application programs which provides a set of interface routines (API) to distribute and retrieve data to/from the disk system. In each I/O node designated, we created a "global" memory cache (file buffer) which caches data that belong to the disk(s) attached to that I/O node (see Figure 6). This cache is implemented as a user level process and shared by all CPUs that use that I/O node (it is also possible to implement it within the Linux kernel). Since multiple CPUs (computation nodes) can share the same memory cache, its efficient utilization is clearly critical. Since global caches have already been studied in the context of PVFS and it is not one of the contributions of this paper, we do not elaborate on our PVFS-based global cache implementation any further in this paper, except for saying that it closely follows the implementation presented in [23]. Our global cache management method employs an LRU (least-recently-used) policy with aging method to determine the best candidate for replacement as a result of a cache miss.

We also implemented the compiler-directed I/O prefetching algorithm explained in Section 3 and our adaptive I/O prefetching scheme, targeting this shared storage cache. We used the SUIF compiler infrastructure [28] to modify the input code for inserting explicit prefetch calls. We observed that the impact of our adaptive prefetch implementation on compilation time was not too much (less than 10% for all four applications used in this work). Also, the code size increase due to the added prefetch calls was less than 17% in these applications. Note that, our approach does not insert any unnecessary prefetch instruction in the code because the insertion of such instructions is based on profiling and compiler analysis. The main reason for increased code size is from the generated helper threads. As given in the example application code in Figure 12, for each loop nest identified as a CPU cluster that exhibits accesses on the shared data, our compiler algorithm generated a separate helper threads for that. Considering the fact that executable sizes of these codes are in hundred kilobyte ranges, we believe that this increase in code size is not that important (in fact, we noticed no increase in the number of instruction cache misses as a result of this increase in executable size).

The experimental results we present in this paper are obtained using a Pentium/Linux based cluster of workstations. Each node of this cluster has a 1.2GHz Intel Pentium-III microprocessor with 32KB of L1 cache, 256KB of L2cache, and 512 MB of main memory. Note that our global cache is implemented on multiple I/O nodes, though most of our results are collected using a single I/O node, and we also present results from a sensitivity analysis that considers multiple I/O nodes, each with its own global cache. Each I/O node is equipped with a 20GB Maxtor hard disk drive, a 32bit PCI10/200Mbps3-Com3c59x network interface card, and a shared cache of 150MB (our default shared storage cache capacity; later we present results with larger caches as well). All the nodes are connected through a Linksys Etherfast 10/200Mbps16 port hub. Our default experimental platform has several computation nodes (the number of which is varied in our experiments) and one I/O node (which implements the global cache).

6. EXPERIMENTAL RESULTS

The performance improvements brought by our adaptive prefetching scheme are presented in Figure 13(a) under the different CPU counts. These improvements are with respect to the no-prefetch case. Comparing this graph with that in Figure 3(a), we see that our approach improves performance significantly. For example, when 8 CPUs are used, the average percentage improvements brought by the independent prefetching scheme and our adaptive prefetching scheme are 9.1% and 19.9%, respectively. More importantly, we observe from this plot that, when our scheme is used, the performance savings obtained using I/O prefetching are quite consistent across different CPU counts. In other words, our approach helps to mitigate the negative impact of harmful I/O prefetches with increasing CPU counts.

At this point, it is also important to compare our scheme to alternate prefetching strategies other than independent I/O prefetching. Figure 13(b) plots, for the 8 and 16 CPU cases, the percentage improvements brought by different I/O prefetching schemes. In this graph, xCPU-Pref denotes a scheme where x CPUs are devoted for prefetching on behalf of the others throughout the entire execution period and the remaining CPUs are used for application execution. We present the results with $1 \leq x \leq 3$, as higher x values generated worse results than those reported in here. Let us first focus on the 8 CPU case. We see that while 1CPU-Pref and 2CPU-Pref produce better savings than independent I/O prefetching, our adaptive scheme results in the best performance among all the schemes tested. Note that fixing the number of CPUs devoted to I/O prefetching at a large value (such as 3 or 4) throughout the entire execution can be dangerous as this can hurt performance in program phases that demand all CPUs for the best result. We can make similar observations in the 16 CPU case as well. In this case however, 3CPU-Perf generated better results as compared to the 8 CPU case since we have a larger number of CPUs to use in executing the application code. In summary, when 8 CPUs are used, our proposed adaptive I/O prefetching scheme improves performance,

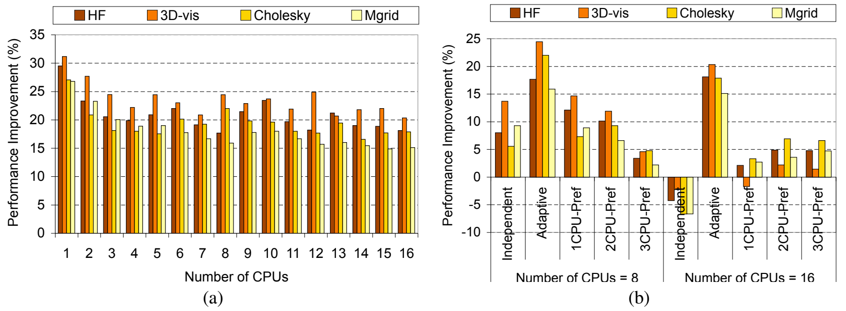

Figure 13: (a) Percentage improvements brought by I/O prefetching when our scheme is used. (b) Comparison of different I/O prefetching schemes.

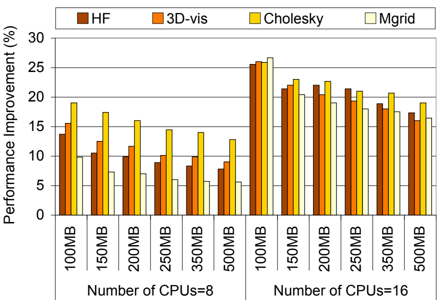

Figure 14: Impact of different storage cache capacities.

on average, by 19.9%, 11.9%, and 10.3% respectively, over the no-prefetching, independent prefetching, and 1CPU-Pref cases. When 16 CPUs are used, the performance improvements over the cases with no prefetching, with independent prefetching, and with 1CPU-Pref cases are 17.9%, 21.7%, and 16.5%, respectively.

6.1 Sensitivity Analysis

In this section, we change the default values of some of our major simulation parameters and conduct a sensitivity analysis. Figure 14 shows, for the 8 and 16 CPU cases, the performance improvements under different shared storage cache capacities. Recall that the default cache capacity used so far was 150MB. Each bar in this graph represents the percentage improvement over the independent I/O prefetching case. Our observation is that, while we witness a reduction in our savings when the cache capacity is increased, even with the largest cache capacity (500MB), we achieve important improvements.

Recall that our experiments so far used only one I/O node. We also performed experiments that measure the sensitivity of our approach to the number of I/O nodes. As mentioned earlier, when multiple I/O nodes are used, we associate a separate global memory cache (of the same size) with each I/O node. The results are presented in Figure 15 with 1, 2 and 4 I/O nodes (the x-axis). Each bar represents the performance improvement brought by our approach over the independent I/O prefetching case. The figure presents the results for only 8 and 16 computation node cases. As expected the percentage savings brought by our approach get reduced when the number of I/O nodes is increased. This is because, with a larger number of I/O nodes, the prefetch requests are spread more and this tends to reduce the number of harmful prefetches. Since the results in Figure 15 are with respect to the case without our optimizations, we observe a drop in percentage savings. Still, even with the largest number of I/O nodes tested, the savings we achieve are not bad.

Recall that so far in our experiments we assigned a common prefetcher to two or more threads if the sharing density is 80% or higher (in other words, the sharing density threshold was 80%). Figure 16 shows the percentage improvement results when the sharing density threshold is varied between 50% and 90%. Our first observation is that when we set the threshold to 90%, 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 CPU ends up with performing its own I/O prefetching. On the other hand, when the threshold is very low (e.g., 50% or 60%), our approach behaves similar to the independent I/O prefetching case.

Finally, 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 close.

7. RELATED WORK

The replacement algorithm for I/O caching has a significant influence on I/O performance. While the LRU (Least Recently Used) replacement policy, which dates back at least to 1965 [10], has been widely used to manage buffer cache, there are various approximations and enhancements to this, for example, the classical CLOCK algorithm [8]. To add adaptability to changing access patterns, several researchers studied enhancements to the classical CLOCK algorithm, such as 2Q [18] and LRFU [9]. More recent studies that try to handle accesses with weak temporal or spatial locality include CAR (Clock with Adaptive Replacement) [4], LIRS (Low Inter-reference Recency Set) [17], ARC (Adaptive Replacement Cache) [24], CLOCK-Pro [29], Second-Tier Cache Management [42], and DULO (Dual LOcality)[30]. Patternson et al [27] used a hint mechanism, which is designed to expose access patterns, in managing prefetching and caching file cache blocks. They also studied the same problems under multi-process execution environments [2]. Dahlin et al [12] on the other hand proposed cooperative caching, in which file caches of many client machines are coordinated to form a more effective global file cache. Kimbrel et al [34] studied the prefetching and caching in a system with parallel disks. [27] also provides a mechanism, called the “prefetch horizon”, to limit prefetches that do not bring any benefit from prefetching. In comparison, our work limits redundantly-issued prefetches based on identified inter-thread data sharing patterns.

I/O prefetching is also a very effective way of improving I/O performance [33, 1, 7, 41, 21, 13]. Mowry et al [33] used compiler-guided information to manage prefetch commands more effectively. They also studied the cases where processes running concurrently generate I/O prefetch commands simultaneously [5]. Li and Shen proposed a memory management scheme that handles non-accessed but prefetched pages separately from the rest of the memory buffer cache [21]. More recent studies to improve conventional I/O prefetching using additional file and access history information include Diskseen [41], Competitive Prefetching [7] and AMP [15]. In comparison to these studies, our work targets multiple-CPU execution scenarios.

Targeting multi-level caches, several multi-level buffer cache management policies have been proposed [43, 40, 23, 14]. [40] introduced a DEMOTE operation where an evicted cache block is migrated to lower level of buffer cache. Chen et al [43] used eviction history observed in a higher level cache in determining cache blocks that need to be replaced in a lower level. Lastly, Yadgar et al [14] proposed an approach, called Karma, that uses application hints in maintaining the multi-level cache hierarchy.

The concept of a single separate helper thread to aid the computation thread by exclusively prefetching the data required by the computation thread has been explored in the domain of CMPs (Chip Multiprocessors). Jung et al [6] use a helper thread based prefetching scheme for loosely-coupled processors, like the modern CMPs, and demonstrate the utility of a helper thread in aiding the computation. Kim et al [20] employ similar helper threads running in

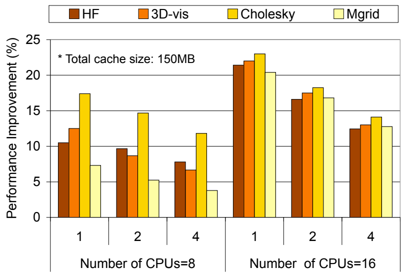

Figure 15: Impact of the number of I/O nodes.

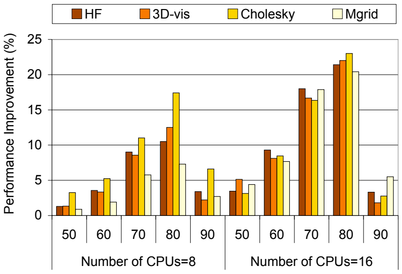

Figure 16: Impact of the sharing density threshold.

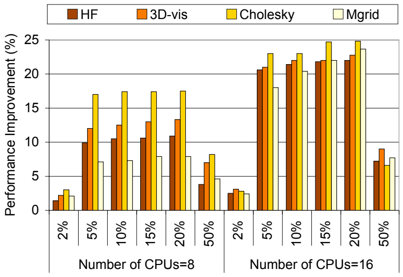

Figure 17: Impact of different slice sizes (S).

spare hardware contexts ahead of the main computation to start the memory operations early so as to hide the memory latency. Liao et al [31] identify and embed trigger points in the original binary and produce a new binary with the prefetch threads attached. Our approach is different from these efforts in two aspects. First, we consider the cases where we have more than two CPUs, and consequently, we employ a different (slice based) code restructuring strategy. To our knowledge, the prior chip multiprocessor/helper thread based efforts target at two-CPU cases. In addition, to our knowledge, none of the prior studies considered an adaptive approach where prefetch threads change based on data sharing during the course of execution. Second, we target I/O intensive applications. We want to say however that our adaptive prefetching algorithm can be used, with appropriate modifications, in a CMP based execution environment as well.

8. CONCLUSION

This paper presents a scheme that extends the concept of compiler-directed I/O prefetching to the multiple CPU case using an adaptive strategy. In the proposed adaptive scheme, the number and types of I/O prefetchers are modulated to match inter-thread data sharing patterns. The experimental results collected using four disk-intensive applications are very promising, and indicate that large performance gains are possible through adaptive I/O prefetching. The percentage improvements brought by our approach over the no-prefetching, independent prefetching, and one-CPU prefetching cases are 19.9%, 11.9%, and 10.3%, on average, when 8 CPUs are used. The average percentage improvements over the same three cases are 17.9%, 21.7%, and 16.5% respectively, when 16 CPUs are used.

9. REFERENCES

[1] A. D. Brown et al. Compiler-Based I/O Prefetching for Out-of-Core Applications. *ACM Trans. Comput. Syst.*, 19(2):111–170, 2001.

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

[3] B. C. Gunter et al. Parallel Out-of-Core Cholesky and QR Factorizations with Pooclapack. In *IPDPS*, pages 1885–1894, 2001.

[4] S. Bansal and D. S. Modha. CAR: Clock with Adaptive Replacement. In *USENIX FAST*, pages 187–200, 2004.

[5] 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.

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

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

[8] F. J. Corbato. A Paging Experiment with the Multics System. Technical Report MIT Project MAC Reort MAC-M-384, May 1968.

[9] D. Lee et al. On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. In *SIGMETRICS*, pages 134–143, 1999.

[10] P. J. Denning. Working Sets Past and Present. *IEEE Trans. Software Eng.*, 6(1):64–84, 1980.

[11] C.-L. C. et al. Improving I/O Response Times via Prefetching and Storage System Reorganization. In *COMPSAC*, pages 143–148, 1997.

[12] M. D. et al. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *OSDI*, pages 267–280, 1994.

[13] P. C. et al. A Study of Integrated Prefetching and Caching Strategies. In *SIGMETRICS*, pages 188–197, 1995.

[14] G. Yadgar et al. Karma: Know-it-All Replacement for a Multilevel Cache. In *USENIX FAST*, pages 25–25, 2007.

[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] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7):28–35, 2000.

[17] S. Jiang and X. Zhang. LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *SIGMETRICS*, pages 31–42, 2002.

[18] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB*, pages 439–450, 1994.

[19] M. Kallahalla and P. J. Varman. Optimal Prefetching and Caching for Parallel I/O Sytems. In *SPAA*, pages 219–228, 2001.

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

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

[22] M. A. Kandaswamy et al. An Experimental Evaluation of I/O Optimizations on Different Applications. *IEEE Trans. Parallel Distrib. Syst.*, 13(12):1303–1319, 2002.

[23] M. Vilayannur et al. Discretionary Caching for I/O on Clusters. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 96–103, 2003.

[24] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *USENIX FAST*, pages 115–130, 2003.

[25] P. H. Carns et al. PVFS: A Parallel File System for Linux Clusters. In *Proc. of the 4th Annual Linux Showcase and Conference*, pages 317–327, 2000.

[26] 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.

[27] R. H. Patterson et al. Informed Prefetching and Caching. In *SOSP*, pages 79–95, 1995.

[28] R. P. Wilson et al. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Not.*, 29(12):31–37, 1994.

[29] S. Jiang et al. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *USENIX*, pages 35–35, 2005.

[30] S. Jiang et al. DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Localities. In *USENIX FAST*, 2005.

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

[32] T. C. Mowry et al. Design and Evaluation of a Compiler Algorithm for Prefetching. In *ASPLOS*, pages 62–73, 1992.

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

[34] T. Kimbrel et al. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *OSDI*, pages 19–34, 1996.

[35] R. Thakur and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. In *Scientific Programming*, pages 301–317, 1996.

[36] N. Tran and D. A. Reed. Automatic ARIMA Time Series Modeling for Adaptive I/O Prefetching. *IEEE Trans. Parallel Distrib. Syst.*, 15(4):362–377, 2004.

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

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

[39] P. Wong and R. F. V. der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, NASA Advanced Supercomputing Division, January 2003.

[40] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *USENIX*, pages 161–175, 2002.

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

[42] X. Li et al. Second-Tier Cache Management using Write Hints. In *USENIX FAST*, pages 115–128, 2005.

[43] Z. Chen et al. Eviction-Based Cache Placement for Storage Caches. In *USENIX*, pages 269–281, 2003.