Prefetch Throttling and Data Pinning for Improving Performance of Shared Caches

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

Seung Woo Son, Mahmut Kandemir
Pennsylvania State University
Email: {sson,kandemir}@cse.psu.edu

Mustafa Karakoy
Imperial College
Email: m.karakoy@ic.ac.uk

***Abstract*—In this paper, we (i) quantify the impact of compiler-directed I/O prefetching on shared caches at I/O nodes. The experimental data collected shows that while I/O prefetching brings some benefits, its effectiveness reduces significantly as the number of clients (compute nodes) is increased; (ii) identify inter-client misses due to harmful I/O prefetches as one of the main sources for this reduction in performance with increased number of clients; and (iii) propose and experimentally evaluate prefetch throttling and data pinning schemes to improve performance of I/O prefetching. Prefetch throttling prevents one or more clients from issuing further prefetches if such prefetches are predicted to be harmful, i.e., replace from the memory cache the useful data accessed by other clients. Data pinning on the other hand makes selected data blocks immune to harmful prefetches by pinning them in the memory cache. We show that these two schemes can be applied in isolation or combined together, and they can be applied at a coarse or fine granularity. Our experiments with these two optimizations using four disk-intensive applications reveal that they can improve performance by 9.7% and 15.1% on average, over standard compiler-directed I/O prefetching and no-prefetch case, respectively, when 8 clients are used.**

I. Introduction

As high-performance applications continuously grow in both size and complexity, there is also a corresponding increase in the sizes of the data sets they process. Most of the data read and written by these applications are disk resident, and therefore, their I/O behavior is a critical factor in determining their overall performance. Unfortunately, most of the I/O parallelism in high-performance applications is dictated by decisions taken during computation parallelization. As a result, I/O parallelism may not be well coordinated across clients (compute nodes), especially when the I/O nodes are shared by multiple clients.

An instance of this situation is I/O prefetching, which is an important optimization for improving performance, as has been demonstrated in the past [27], [31], [25], [32], [18], [5]. In I/O prefetching, data is brought from the disk to the memory cache ahead of time to hide the latency of the disk access. However, I/O prefetching is known to be very sensitive to timing [25]. First, an early prefetch may not be very useful as the data block brought (prefetched) 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 latency. Third, a prefetched data can be even harmful by kicking out a data block from the 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 clients, as shown in Figure 1), this type of "harmful prefetches" can involve different clients as well. For example, a prefetched data block can displace a data block which would be accessed earlier (by possibly another client) than the prefetched data block. We can distinguish between two types of such harmful I/O prefetches. An intra-client harmful prefetch is said to occur when the prefetched data block replaces a block that will be used by the prefetching node. By comparison, an inter-client harmful prefetching occurs when the prefetched block replaces a data block that will be used (before the prefetched block being accessed) by another client.

Clearly, the number of harmful prefetches can increase with the increased number of clients, and consequently, one can expect the harmful prefetch 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 applications that process disk-resident data sets and software (compiler-directed) prefetching, and proposes two solutions – which can be used in isolation or together – to address it. We call our first solution "prefetch throttling" as it controls the number of prefetches issued to the disk system. Our second solution, referred to as "data pinning," prevents selected data blocks from being removed from the shared cache as a result of prefetch operations.

We can summarize the major contributions of this work as follows:

• We quantify the impact of conventional compiler-directed I/O prefetching on shared caches at I/O nodes. The experimental data collected using four high-performance applications that use disk-resident data sets reveals that, while conventional I/O prefetching brings certain performance benefits, its effectiveness reduces significantly as the number of clients is increased.

• We identify inter-client misses originating from harmful prefetches as one of the important sources for this reduction in I/O prefetching performance with increased number of clients.

• We propose and experimentally evaluate two novel schemes, "prefetch throttling" and "data pinning", to improve performance of I/O prefetching. Prefetch throttling prevents one or more clients from issuing further prefetches if such prefetches are predicted to be harmful, i.e., replace from the memory cache the useful data accessed by other clients before the prefetched data being accessed. Data pinning on the other hand makes selected data blocks immune to harmful prefetches by pinning them in the memory cache. We show that these two optimizations can be applied in isolation or can be combined together. In addition, they can be applied at a coarse or fine granularity (in this work we evaluate both these options).

The proposed I/O optimization schemes can be implemented at the file system level or at the disk controller level. Our experiments with a file system based implementation (using PVFS [4]) on a Linux cluster of these two schemes reveal that they can improve performance (overall execution time) by 9.7% and 15.1% on average, over standard compiler-directed I/O prefetching and no-prefetch case, respectively,

SC2008 November 2008, Austin, Texas, USA 978-1-4244-2835-9/08 $25.00 ©2008 IEEE

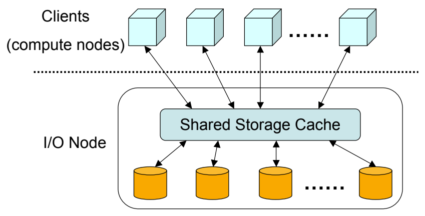


Fig. 1. I/O architecture.

when 8 clients are used. In addition, the experimental results show that our schemes achieve consistent savings under the different values of our major experimental parameters, and the performance improvements achieved come very close to those that can be obtained using a hypothetical optimal prefetching scheme.

The rest of this paper is structured as follows. The next section gives the descriptions of the I/O architecture considered in this work and of the baseline compiler-directed I/O prefetching scheme. Sections III and IV present, respectively, the experimental platform we used and the results collected using I/O prefetching under the different client counts. Our proposed prefetch throttling and data pinning schemes are explained and experimentally evaluated in Section V. Section VI presents results from our sensitivity experiments, summarizes several enhancements over our baseline implementation of prefetch throttling and data pinning, and discusses possible future directions. Section VII discusses related work, and Section VIII concludes the paper by summarizing our main contributions and future directions.

II. Compiler-Directed I/O Prefetching

Figure 1 depicts the architecture targeted by our work. In this architecture, an I/O node is "shared" by multiple clients (compute nodes) and contains a shared storage cache that serves these clients. This shared cache presents, from a performance perspective, opportunities (e.g., fast data sharing across different clients through memory) as well as potential drawbacks (e.g., destructive interferences among data streams of different clients). Our focus in this work is on quantifying the impact of I/O prefetching targeting this shared memory cache and is to propose schemes to address the harmful prefetch problem. While most of our experiments use a single I/O node based configuration (with a single disk attached), we also report results with multiple I/O node configurations, each with its own storage cache shared possibly by a subset of available clients (see our sensitivity experiments in Section VI). In the rest of this paper, when no confusion occurs, we use the terms "client", "processor", and "compute node", interchangeably.

While there exist several I/O prefetching algorithms published in literature [27], [31], [25], [32], [18], [5], the one used in this work is inspired by the work done by Mowry et al [25]. The original algorithm has actually been proposed for improving hardware cache behavior for memory-resident data sets [26], 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. In [25], prefetches are inserted into the code based on a data reuse analysis. More specifically, an optimizing compiler analyzes the application code and identifies future data elements that need to be brought to the memory cache. It then inserts explicit prefetch instructions to bring such

```
int $U1[N1 \times N2]$;
int $U2[N1 \times N2]$;
int $U3[N1 \times N2]$;
...
for $i = 1$ to $N1$ {
  for $j = 1$ to $N2$ {
    $U1[i,j]$ = $U2[i,j]$ + $\alpha$*($U3[i,j]$-2.0*$U2[i,j]$ + $U1[i,j]$);
    $U2[i,j]$ = $U3[i,j]$;
  }
}
```

(a)

```
int $U1[N1 \times N2]$;
int $U2[N1 \times N2]$;
int $U3[N1 \times N2]$;
...
for $i = 1$ to $N1$ {
  prefetch (&$U1[i][0]$, $B$);          /* prolog */
  prefetch (&$U2[i][0]$, $B$);
  prefetch (&$U3[i][0]$, $B$);
  for $jj = 1$ to $N2 - X$, $X$ {       /* steady state */
    prefetch (&$U1[i][jj+B]$, $B$);
    prefetch (&$U2[i][jj+B]$, $B$);
    prefetch (&$U3[i][jj+B]$, $B$);
    for $j = jj$ to $jj + X$ {
      $U1[i,j]$ = $U2[i,j]$ + $\alpha$*($U3[i,j]$-2.0*$U2[i,j]$ + $U1[i,j]$);
      $U2[i,j]$ = $U3[i,j]$;
    }
  }
  for $j = N2 - X$ to $N2$ {            /* epilog */
    $U1[i,j]$ = $U2[i,j]$ + $\alpha$*($U3[i,j]$-2.0*$U2[i,j]$ + $U1[i,j]$);
    $U2[i,j]$ = $U3[i,j]$;
  }
}
```

(b)

Fig. 2. An example that illustrates compiler-directed I/O prefetching. (a) Original code fragment. (b) Compiler-generated code with explicit I/O prefetch calls inserted.

elements into the cache ahead of time to ensure that data is in the cache when it is actually referenced.

Figure 2 illustrates an example application of this compiler-directed I/O prefetching scheme. For the sake of illustration, we omit the actual file I/O statements. In this example, three disk-resident arrays ($U1$, $U2$, and $U3$) of the same size ($N1 \times N2$) are accessed by two statements in the innermost loop body. Figure 2(a) shows the original loop (without any I/O prefetching), and Figure 2(b) illustrates the compiler-generated code after the I/O prefetch calls are embedded by the compiler. In this example, B is assumed to be the unit of I/O prefetching, i.e., when an I/O prefetch instruction with argument B is called, it brings B data elements from the disk. In a virtual memory based environment, this B parameter is typically chosen such that it corresponds to the page size of the underlying platform. Based on this unit of prefetch and the I/O latency associated with it, expressed in terms of the number of cycles, the number of iterations, X, ahead of which data should be prefetched, can be given by:

$$X = \lceil \frac{T_p}{s + T_i} \rceil,$$

where X is the prefetch distance, T_p is the I/O latency to prefetch B blocks, s is the number of iterations in the shortest path through the loop body, and T_i is the overhead incurred by a prefetch call inserted. Note that, in order to make prefetch points explicit with respect to the specified block size (B), a given loop nest may have to be decomposed into two loop nests. As can be seen in Figure 2(b), the inner loop j, which is selected for inserting prefetch instructions, is transformed into two loops: the outer loop (the strip loop) and the inner loop (the element loop). This particular transformation is called strip-mining [35] and, in this example, we assume that the outer loop has a strip size of X, as calculated above. As a result of this, the jj loop iterates over individual data blocks, whereas the j loop iterates over the elements within a block. The prefetch instructions are inserted before starting the innermost loop, thereby prefetching a data block before operating on the data elements it contains. Following this, the actual computation of the data elements in the block is performed on the blocks prefetched previously. The epilog iterations are executed by the last loop shown in the figure.

We now discuss the compiler analysis required for implementing this I/O prefetching. First, the compiler analyzes the given application code[1] and predicts the future data access patterns. This is done using data reuse analysis, a technique developed originally for conventional cache locality optimization [19]. 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) of different data elements. Note that several prefetching techniques have been proposed in literature to hide the memory-to-cache latencies. These techniques typically try to reduce the number of unnecessary prefetches due to data elements that are already in the cache. This is also important in the case of I/O prefetching as we do not want to prefetch a data element that is already in the memory cache (e.g., for each data block, we need to issue a prefetch request for only the first element). In [25], 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 [25]), hence inner loop nests often access less data than a page size. In deciding the loop splitting point, the prefetching algorithm takes into account 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. 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 actually the case, that prefetch is suppressed. In this way, a significant number of useless prefetches can be eliminated.

We also 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 prefetch throttling and data pinning schemes can be used along with any existing I/O prefetching algorithm. Clearly, the savings achieved by our schemes are 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 [25] inserts I/O prefetches very carefully taking into account loop-specific I/O behavior and estimated I/O latencies. As a result, it inserts very few useless prefetches and most of such useless prefetches are filtered before they are actually issued to the disk. Simpler schemes on the other hand tend to insert more useless prefetches (some of which will also be harmful prefetches), and we can expect this to further increase the effectiveness of our throttling and pinning schemes. In fact, our experiments with an alternate (simpler) I/O prefetching scheme indicate that the proposed schemes achieve significantly higher savings as compared to those obtained with the compiler-directed prefetching scheme (these results will be presented later in Section VI).

[1]In our case, this input code already contains explicit I/O (file read and file write) calls. The compiler-directed prefetching algorithm augments this input code with explicit prefetch calls.

III. Experimental Platform and Benchmarks

In order to test the effectiveness of our proposed schemes and conventional I/O prefetching, we performed experiments with four applications that process disk resident data sets:

• *mgrid:* This application demonstrates the capabilities of a simple multigrid solver in computing a three dimensional potential field. The original application, which is part of both the NAS [1] and SPEC [13] benchmark suites is re-coded, for the purposes of this study, to perform explicit I/O from disk. Data sets are made disk resident and additional optimizations such as collective I/O [29] have been inserted to maximize the I/O performance as much as possible. In this application, in addition to echoing some of the inputs, the main part of the output gives the smoothed approximate inverse. In a typical run, the total size of the data manipulated by this application is about 9.3GB.

• *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 [12] and sub-portions of the main disk-resident matrix are transferred to memory as needed. As in the case of mgrid, the I/O behavior of the application has been carefully optimized as much as possible using known techniques such as collective I/O [29]. The total size of the data manipulated by this benchmark is about 11.7GB in our single I/O node configuration.

• *neighbor_m:* This is a data mining algorithm that uses a nearest neighbor method [20]. This method, by maintaining a dataset of known records, finds records (neighbors) similar to a target record and uses the neighbors for classification and prediction. The particular application we have implements market basket analysis, and the amount of data processed is over 16GB in our default configuration. This application heavily uses an I/O optimization called data sieving [30].

• *med:* This is an advanced image processing and measurement software for MRI. It processes 3D images and reslices them along multiple axes. It also has a module which combines multi-modality images to create image fusions. This application uses both data sieving [30] and collective I/O [29]. In a typical run, the total disk resident data processed by this application is around 14GB.

We made our experiments using PVFS, the Parallel Virtual File System [4], which runs on top of a Linux cluster. PVFS is a mainly user-level implementation, i.e., there is a library

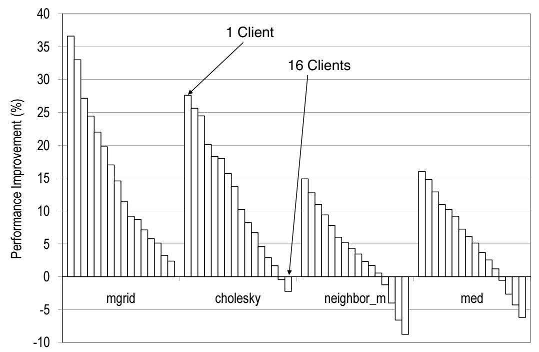


Fig. 3. Percentage improvements in total execution cycles due to I/O prefetching (over the no-prefetch case). In each application, the bars from left to right are for 1 through 16 clients.

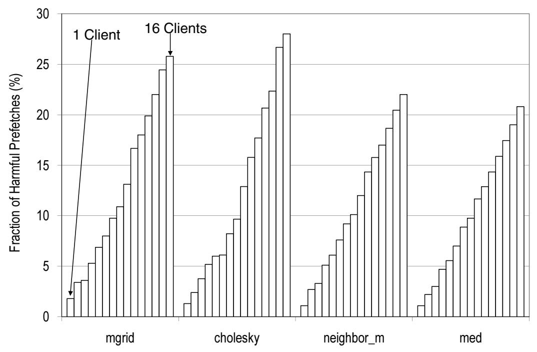


Fig. 4. Fraction of harmful prefetches. In each application, the bars from left to right are for 1 through 16 clients.

(libpvfs) linked 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 which caches data that belong to the disk(s) attached to that I/O node. This cache is implemented as a user level process and shared by all compute nodes (clients) that use that I/O node (it is also possible to implement it within the Linux kernel). Since multiple clients share the same memory cache, its efficient utilization is clearly very 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 global cache implementation any further in this paper, except for saying that it closely follows the implementation presented in [33]. Our global cache management method employs a LRU (least-recently-used) policy with aging method to determine a best candidate for replacement as a result of a cache miss.

We also implemented the compiler-directed I/O prefetching algorithm explained in Section II targeting this global cache. We used the SUIF compiler infrastructure [34] to modify the input code for inserting explicit I/O prefetch calls. SUIF is an optimizing compiler infrastructure that can be used as a source-to-source translator. It consists of a small kernel and a suite of compiler passes built on top of the kernel. The SUIF kernel defines the intermediate representation, provides functions to access and manipulate the intermediate representation, and structures the interface between successive compiler passes. Our prefetch insertion algorithm is implemented as a standalone phase within SUIF. We observed that the impact of our prefetch implementation on compilation time was not too much (less than 20% for the four applications used in this work). Also, the code size increase due to added prefetch calls was less than 18% for our applications. Considering the fact that executable sizes for 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 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 (which can be configured as I/O and/or compute node) of this cluster has a 800 MHz Intel Pentium processor with 32KB of L1 cache, 256KB of L2 cache, and 2GB of physical main memory. In our default configuration, we allocated 256MB of this memory as our default cache for I/O. Note that, as stated earlier, our global cache is implemented on multiple I/O nodes, though most of our results are collected using a single I/O node. 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 and a network interface card. All the nodes are connected through a Linksys Etherfast 10/100Mbps 16 port hub. Our default experimental platform has several compute nodes (the number of which is varied in our experiments) and one I/O node (which implements the global cache). Unless stated otherwise, each compute node (client) has also a client side cache of 64MB.

IV. EVALUATION OF COMPILER-DIRECTED I/O PREFETCHING

Figure 3 presents the "percentage improvements in total execution cycles" of our four data-intensive applications due to I/O prefetching, under the different number of clients. Specifically, each bar corresponds to the performance improvement brought by the prefetching scheme in [25] over the *no-prefetch case*. An important observation from these results is that the effectiveness of prefetching dramatically diminishes as the number of clients to execute the application code increases. For example, with mgrid, the improvement brought by prefetching is about 36.6% when the single client is used, whereas the same is only 2.3% with 16 clients. In fact, I/O prefetching degrades overall performance in cholesky, neighbor_m, and med, when 15 or 16 clients are used, and in the last two applications even with 13 or 14 clients. To understand why this happens, we collected additional statistics capturing the prefetch-related interactions among the clients. Our first group of statistics are presented in Figure 4 and gives the fraction of harmful prefetches. As stated earlier, we define a "harmful prefetch" as a prefetch that leads to the removal of a data block from the cache and the prefetched data block is referenced only after the reference to the removed block. We see from Figure 4 that, the contribution of harmful prefetches increases with the increasing number of clients. This in a sense can be expected, as more clients are used for executing the application, higher the chances that clients will replace each other's data from the 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 clients. For example, we also noticed during our experiments that the negative interactions even among normal disk fetches to the memory cache tend to increase with the large number of clients. Nevertheless, the results presented in Figures 3 and 4 illustrate a strong correlation between the degradation in the effectiveness of I/O

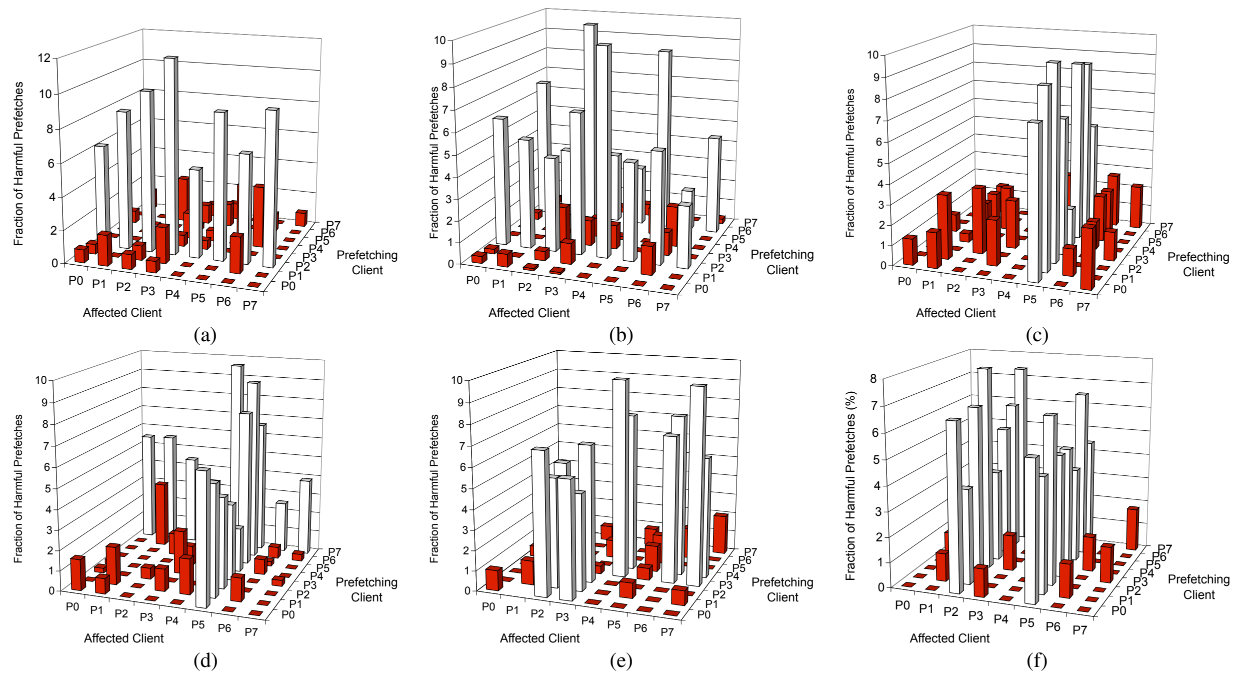


Fig. 5. Statistics collected at different points during the course of execution of our four applications. Each bar-chart shows a distribution of the harmful I/O prefetches for the execution that uses 8 clients. (a) and (b) are from mgrid; (c) is from neighbor_m; (d) and (e) are from cholesky; and (f) is from med. The white bars indicate interesting patterns.

prefetching and the fraction of harmful I/O prefetches, which deserves further investigation.

As stated earlier, harmful prefetches can be either "intra-client" (when the prefetched data block replaces a block that will be used by the prefetching client) or "inter-client" (when the prefetched block replaces a data block that will be used by another client). In the first case, the prefetch discards the data used by the same client, whereas in the second case the prefetching client and the client that makes reference to the discarded data are different. Figure 5 shows some statistics collected at different points during the course of execution of our applications. Each bar-chart in this figure shows a distribution of the harmful prefetches for an execution that uses 8 clients (P0 through P7). In each bar-chart, "Prefetching client" is the client that performs I/O prefetching and "Affected client" is the client whose data got removed from the shared cache as a result of this I/O prefetching (i.e., whose data is the victim of a harmful prefetch). We want to emphasize that only harmful prefetches are included in these graphs. To collect these statistics, the application execution is divided into 100 "epochs," and the bar-charts shown illustrate interesting and representative patterns extracted from the collected statistics. As the execution moves from one epoch to the next, we reset the counters used to collect the statistics to capture the new dynamic behavior.

Let us start by studying the graph in (a), which is captured during one of the initial epochs of mgrid. We see that the most of the harmful prefetches (more than 66%) are the ones issued by the second client P2. The graph in (b), which represents an epoch toward the middle of execution of the same application, exhibits a different trend than the one in (a). In this case, two clients (P2 and P6) are responsible for a large majority of harmful prefetches (more than 85%). The third pattern – shown in (c) – is taken from one of the last epochs of neighbor_m, and demonstrates an entirely different behavior than the previous two. In particular, here we observe that one of the clients (P5) is the victim of most of the harmful prefetches. The next two bar-charts in Figure 5 are taken from cholesky and capture two representative behaviors (one from the beginning of the application execution and one towards the end). In (d), we observe two interesting patterns. First, most of the harmful prefetches are issued by one of the clients (P7), and second, among all clients, P5 is the one that is affected most by the harmful I/O prefetches. The graph in (e) indicates a more clustered behavior. Specifically, there are a few clients which perform harmful prefetches that affect another group of clients. Also, there is another group of clients that are affected greatly by harmful I/O prefetches. Finally, the last bar-chart (f) is taken from the execution of med and shows that two clients (P2 and P5) are affected from most of the harmful prefetches. We need to mention at this point that the patterns shown in Figure 5 are not isolated, infrequently-occurring patterns. They in fact occur very frequently during the course of execution. For example, the first 13 epochs in the beginning of the execution of mgrid exhibit similar pattern to the one shown in (a). Similarly, 8 consecutive patterns in cholesky are very similar to the one given in (d), and med has many patterns similar to that shown in (f). Therefore, if one could take advantage of these patterns during execution, significant performance gains can be achieved. In the rest of this paper, we present two optimization schemes that exploit these harmful prefetch patterns, and quantify the performance benefits they bring in our applications.

V. Our Schemes

A. Prefetch Throttling and Data Pinning

Based on the discussion in the previous section, we propose two schemes for improving the behavior of I/O prefetching:

"prefetch throttling" and "data pinning." In this section, we explain the details of these two optimization schemes and quantify the benefits they bring. Both these schemes are "history based," that is, the execution of the application is divided into epochs and the observations made during the execution of the current epoch are used to optimize the behavior of the next epoch.

In prefetch throttling, one or more clients (processors) are (temporarily) prevented from issuing prefetch requests to reduce the number of harmful prefetches. Note that this optimization can be very useful in scenarios such as those depicted in Figures 5(a) and (b). Specifically, we could improve performance of I/O prefetching in the scenario depicted in Figure 5(a) by throttling the I/O prefetches issued by P2, and similarly, in Figure 5(b) by throttling the prefetches of P2 and P6. To implement this scheme, we keep track of the harmful prefetches issued by each client. Specifically, when a data block is prefetched into the shared cache, we record the block it discards, and then later check whether the prefetched block or the discarded block is accessed first. If it is the latter, we increase the counter (which counts the number of harmful prefetches) attached to the prefetching client by one. In addition to these client-local harmful prefetch counters, we also keep track of the total number of harmful prefetches using a global counter. At the end of each epoch, the contents of the local counters and the global counter are used for calculating the individual contribution of each client to the total number of harmful prefetches. The clients whose contributions to harmful prefetches are above a pre-set "threshold value" are prevented from issuing further I/O prefetches in the next epoch. In addition, the counters (including the global one) are reset to 0 before the next epoch starts to ensure that we capture the dynamic variations in the behavior of the application. Our default threshold value – determined based on some preliminary experiments we conducted – is 0.35 which means that if, in epoch e, 35% of the prefetches issued by a client are harmful prefetches, that client is prevented from I/O prefetching in epoch e+1. In our default implementation, this client is automatically enabled to perform I/O prefetches in epoch e+2 (i.e., it resumes its normal prefetches) since it does not perform any prefetches (harmful or not) in epoch e+1. The pseudo-code that explains the operation of prefetch throttling is given in Figure 6.

```
for epoch $e = 1$ to $E$ {
  harmful-prefetches[e]=0;
  for $i = 1$ to $N$ (number of accesses) {
    if (data-access[e][i] is a prefetch and
        prefetch-allowed[i][e]=1) {
      processor=data-access[e][i].proc;
      accessed=data-access[e][i].accessed;
      discarded=data-access[e][i].discarded;
      if (discarded==accessed){
        processor-prefetch-counter[processor]++;
        harmful-prefetches[e]++;
      }
    }
  }
  for i=1 to P (number of processors) {
    if (processor-prefetch-counter[i] $\geq T$) {
      prefetch-allowed[i][e+1]=0;
    }
    processor-prefetch-counter[i]=0;
  }
}
```

Fig. 6. Sketch of our prefetch throttling scheme.

```
for epoch $e = 1$ to $E$ {
  harmful-prefetch-misses[e]=0;
  for $i = 1$ to $N$ (number of accesses) {
    if (data-access[e][i] causes a miss) {
      if (data-access[e][i] is a prefetch) {
        processor=data-access[e][i].proc;
        processor-miss-counter[processor]++;
      }
      misses[e]++;
    }
  }
  for $i = 1$ to $P$ (number of processors) {
    if ((processor-miss-counter[i] / misses[e]) $\geq T$) {
      for all blocks $b$ brought by $i$
        pin[b][e+1]=1;
      }
    }
    processor-miss-counter[i]=0;
  }
}
```

Fig. 7. Sketch of our data pinning scheme.

In data pinning, selected data blocks brought to the memory cache by a client are marked as non-removable (i.e., pinned in the cache) for a certain period of time. This helps those clients which are affected from harmful prefetches significantly. Consider for example the scenario illustrated in Figure 5(c). In this scenario, client P5 suffers a lot from harmful prefetches, and as a result, it can benefit significantly if its data could be protected against harmful prefetches through pinning. Similarly, in Figure 5(f), two clients (P2 and P5) are significantly affected from harmful prefetches and can potentially benefit from data pinning. We implement data pinning using a pre-set "threshold value," similar to the one used in the case of throttling. This time, however, we keep track of – for each client – the fraction of memory cache misses it incurs because of harmful prefetches. This is done by employing a counter for each client that records the number of cache misses it incurs due to harmful prefetches and another counter which keeps track of all misses (across all client) due to harmful prefetches. When, for a given client, the fraction of misses due to harmful prefetches exceeds the threshold value (which is 35% in our experiments) in epoch e, the data blocks brought by that client to the memory cache are pinned in the shared cache during the entire epoch e+1. In this case, when a prefetch tries to kick out a data block of this processor, another victim (from another client) is selected, again based on the LRU policy. In other words, the victim is the block that has not been brought into the cache by that client and has the lowest LRU value among all such blocks. As before, the counter values are reset to 0 at the beginning of each epoch. Note also that the blocks brought into the shared cache in epoch e+2 (by the client whose blocks are pinned in epoch e+1) are not pinned. The pseudo-code that explains the operation of our pinning approach is given in Figure 7.

While both throttling and pinning bring certain overheads such as updating counters and making comparisons at the end of each epoch, we found that these overheads are not excessive in practice, as compared to the large I/O latencies incurred during disk read and writes. Also, some of these overheads occur only at the epoch boundaries. Nevertheless, the results presented in the rest of this paper include "all" the performance overheads incurred by our two optimization schemes.

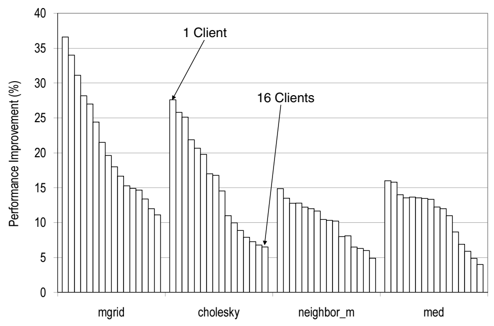


Fig. 8. Percentage improvements in execution cycles when our two optimization schemes are used with I/O prefetching (coarse grain version).

TABLE I
CONTRIBUTIONS OF OVERHEADS TO THE OVERALL APPLICATION EXECUTION TIME. (I) REPRESENTS THE OVERHEADS INVOLVED IN DETECTING HARMFUL PREFETCHES AND CACHE MISSES DUE TO THEM AND UPDATING RELEVANT COUNTERS AND (II) REPRESENTS THE OVERHEADS INVOLVED IN CALCULATING THE FRACTION OF HARMFUL PREFETCHES AND MISSES FOR EACH CLIENT. THE RESULTS ARE GIVEN FOR CLIENT COUNTS OF 2, 4, 8, AND 16.

Benchmark	**2**		**4**		**8**		**16**	
	i	**ii**	**i**	**ii**	**i**	**ii**	**i**	**ii**
mgrid	2.20%	1.81%	3.44%	2.76%	4.16%	3.55%	4.96%	3.97%
cholesky	1.88%	1.26%	2.62%	1.73%	3.27%	2.58%	4.72%	3.36%
neighbor_m	1.97%	1.53%	1.87%	2.08%	3.66%	3.27%	4.14%	3.46%
med	2.13%	1.90%	3.57%	2.92%	3.81%	3.29%	4.73%	3.77%

B. Results

The performance improvements brought by I/O prefetching supported by our throttling and pinning schemes (over the no-prefetch case) are presented in Figure 8 under the different client counts. Comparing this graph with that in Figure 3, we see that our approach improves performance significantly. For example, when 8 clients are used, the percentage improvements brought by our approach over the no-prefetch case are 19.6%, 16.7%, 10.4% and 13.3% for mgrid, cholesky, neighbor_m, and med, respectively. Note that these results are much better than the corresponding improvements achieved by the straightforward I/O prefetching scheme, which are 14.5%, 13.7%, 4.3% and 6.1%, respectively. Table I gives the contributions of the overheads (incurred by our schemes) to the overall application execution time. As stated above, the results presented in this paper include all the overheads incurred by our optimization schemes. Basically, there are two major sources of overhead in our schemes: (i) detecting harmful prefetches and cache misses due to them and updating relevant counters (which is performed at every cache miss and prefetch) and (ii) calculating the fraction of harmful prefetches and misses of each client (which is performed at the end of every epoch). Our observation from Table I is that the total contribution of these overheads to the overall execution latency is less than 9% for our benchmarks. We also see that contribution of (i) is larger than (ii), mainly because the former takes place much more frequently than the latter.

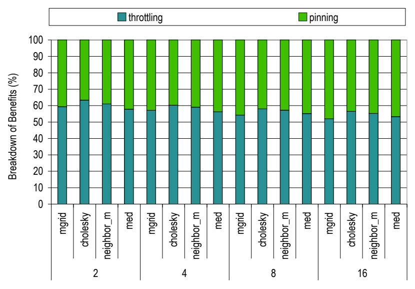
Before moving to the explanation and evaluation of the fine grain version of our approach, we present the breakdown of the performance benefits brought by our approach. In Figure 9(a), each bar represents the total performance benefits brought by prefetch throttling and data pinning, and is set to 100. The lower portion of a bar gives the percentage benefits brought by prefetch throttling alone, while the upper portion captures the performance benefits obtained through data pinning alone. We see that, while prefetch throttling is in general (but not always) more beneficial than data pinning, the relative contribution of pinning increases as we increase the number of clients. This is because, for a given data block, the chances for it to get displaced as a result of prefetching increases as we increase the number of clients.

C. Fine-Grain Optimization

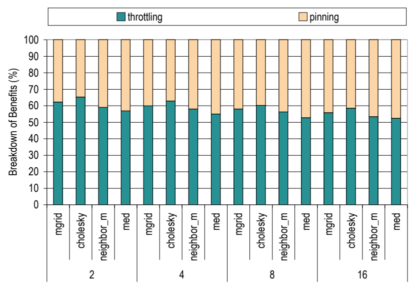
The optimizations explained thus far in this section can be considered “coarse grain” as they keep track of harmful prefetches and misses due to them from the individual client’s perspective rather than in a client-pair centric manner. In other words, if the prefetches of a client are throttled, some useful prefetches issued by that client will also be throttled, and this can affect performance negatively. Similarly, when data blocks of a client are pinned, they are pinned against all prefetches from all clients. Some of these prefetches may in fact be useful (i.e., have nothing to do with the protected blocks) and can improve performance if enabled. In this section, we evaluate a “fine grain” version of our approach. In this version, which is oriented to address the problems associated with the coarse grain version explained above, we keep track of harmful prefetches and misses due to harmful prefetches for each client pair. In a sense, we try to capture – at runtime – the information represented in bar-charts of the type shown in Figure 5. Note that this level of information can allow us to perform detailed optimizations which could not be possible under the coarse grain implementation. For example, by keeping harmful prefetch and cache miss information at a client pair granularity, we can throttle I/O prefetches from a client only if such prefetches will discard the data of a certain other client or set of clients, and let the remaining prefetches it issues go through. In other words, it can be possible to tune the behavior of each client – as far as I/O prefetching is concerned – with respect to the behavior of every other client individually.

To make this point clear, let us consider the bar-chart in Figure 5(e). Unlike the graphs given in Figures 5(a) through (d) and (f), we observe a “clustered-behavior” in this graph. For example, we can see that the prefetches performed by clients P0, P1 and P2 most affect the data of clients P1, P2 and P3. Similarly, prefetches carried out by P3 and P4 displace from the cache mostly the data required by P4, P6 and P7. By taking into account this finer grain pattern of harmful prefetches, one can potentially do a much better job than the coarse grain version evaluated so far. For example, instead of throttling all prefetches from client P0, one can throttle only the ones that will displace data of clients P1, P2 or P3. A similar fine grain approach can be used for pinning as well. As an example, instead of pinning the data blocks of client P3 against all prefetches, we can pin them only against prefetches from clients P0, P1 and P2.

Implementing this finer grain version of our approach requires us maintain more counters during execution. As an example, for a system with p clients, we need $p^2 + 1$ counters for keeping track of the harmful prefetches at a client pair granularity to implement the finer grain version of our throttling scheme. p^2 of these counters are for the client pairs, and the last one is a global counter that keeps track of all



(a)



(b)

Fig. 9. Breakdown of the percentage benefits brought by our schemes. (a) coarse grain optimization. (b) fine grain optimization. The results are given for client counts of 2, 4, 8 and 16 (on the x-axis).

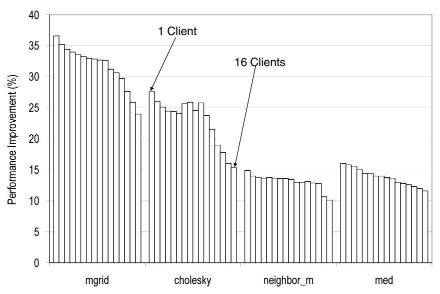


Fig. 10. Percentage improvements in execution cycles when our two optimization schemes are used with I/O prefetching (fine grain version).

harmful prefetches. A similar number of counters are also needed for implementing the fine grain version of our data pinning scheme. However, we noticed during our experiments that the total performance overhead brought by maintaining these counters and making throttling/pinning decisions was below 12% for our applications (which is slightly larger than the 9% overhead incurred when using the coarse grain version). As in the case of the coarse grain version, this fine grain version also needs to use threshold values to decide when the interaction between two clients is harmful enough. In prefetch throttling, when the fraction of (total) harmful prefetches issued by client P_k and affect client P_l exceeds a certain pre-set threshold value, the prefetches of P_k (in the next epoch) that are designated to displace a block of P_l are throttled. Similarly, in deciding to pin certain blocks of client P_k against prefetches issued by client P_l, we use a threshold value. In our default implementation of the fine grain version, both these threshold values are set to 20% based on some preliminary experiments. Later, we also present experimental results with different threshold values.

The results (percentage improvements in application execution times over the no-prefetch case) with this fine grain version of our approach are presented in Figure 10. When we compare these results with the corresponding ones obtained using the coarse grain version (presented in Figure 8), we see significant improvements. For example, when using 8 clients the savings with the fine grain version are about 34.6% and 25.9% for mgrid and cholesky respectively, which are much better than the corresponding numbers obtained through the coarse grain version (19.6% for mgrid and 16.8% for cholesky). Figure 9(b) presents the contributions of throttling and pinning in this fine grain version. As in the case of the coarse grain version, we see that both throttling and pinning contribute to the performance savings achieved.

VI. Discussion

In this section, we first change the default values of some of our major simulation parameters and conduct a sensitivity analysis. We then evaluate the influence of our schemes when they are used with a simpler (runtime behavior based) I/O prefetching scheme, instead of the sophisticated compiler-based prefetching used in our experiments so far. After that, we briefly discuss and evaluate a variant of our approach that takes into account stable behavior (patterns), from the perspective of harmful prefetch patterns, of consecutive epochs during execution. Then, we present the scalability of the proposed approach and how it performs under multiple application scenario. After giving a comparison to our approach to an optimal scheme, we discuss future directions.

Sensitivity to Default Parameters. Recall that our experiments so far used only one I/O node with cache size of 256MB. We also performed experiments that measure the sensitivity of our approach to the number of I/O nodes while the total cache size is kept at 256MB. 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 11 with 2, 4, and 8 I/O nodes. The figure presents the results for only 8 and 16 client cases with the fine grain version. 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 and this tends to reduce the number of harmful prefetches. Since the results in Figure 11 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 performance improvements we achieve are not bad.

Our next set of experiments measure the performance benefits brought by our approach when the buffer (shared cache) size in an I/O node is varied. Recall that the total shared cache size used in our experiments so far is 256MB, which is the memory space allocated for buffer cache within a single I/O node. Figure 12 shows the results with cache sizes of 128MB, 256MB, 512MB, 1GB, and 2GB when the fine-grain version is applied (single I/O node). As stated in Section III, the default cache space allocated within a single I/O node is 256MB and we changed it to different cache sizes. Our observation is that, while our savings get reduced with larger buffer sizes, we still achieve significant performance improvements. For instance, for the 16 client case with a 1GB shared cache space, the average performance improvement when all four applications are considered is about 9.5%. These results also indicate that we can expect our approach to be even more effective as the data sizes manipulated by I/O-intensive applications keep increasing.

Figure 13 gives the performance improvements achieved by the fine grain version of our scheme in more detail for the case with 2GB buffer size. As in Figure 12, we used a single I/O node, which is configured with 2GB buffer cache. We observe that, for all client counts tested, our approach generates reasonable savings (even with this large buffer capacity). Overall, when we consider the results given in

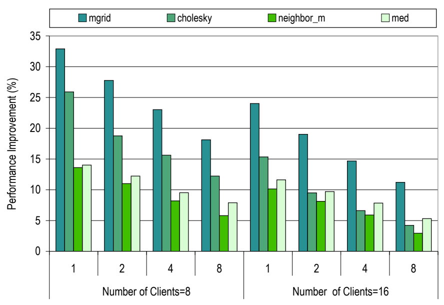


Fig. 11. Percentage savings with the I/O node counts of 1, 2, 4, and 8 (on the x-axis). Results are given for the 8 and 16 client cases under the fine grain version.

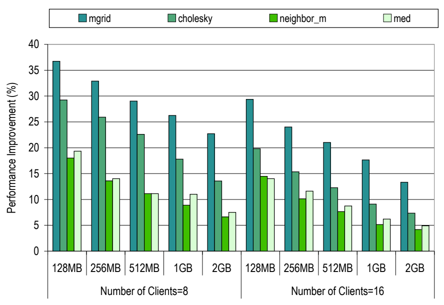


Fig. 12. Percentage savings with the different buffer sizes (on the x-axis). Results are given for the 8 and 16 client cases under the fine grain version.

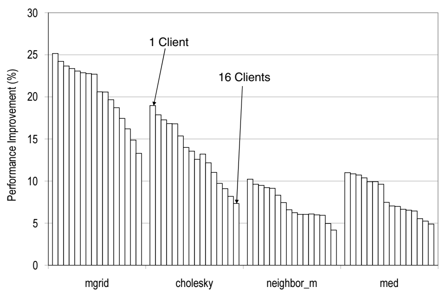


Fig. 13. Percentage improvements with 2GB buffer size.

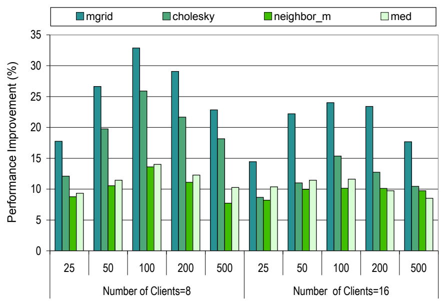


Fig. 14. Percentage savings when the number of epochs (on the x-axis) is varied.

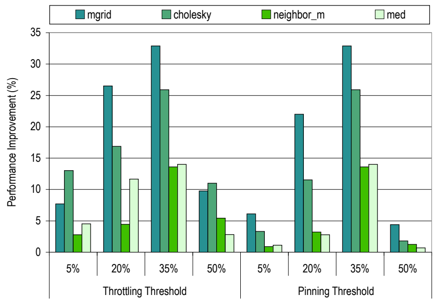


Fig. 15. Percentage savings with the different threshold values (the x-axis).

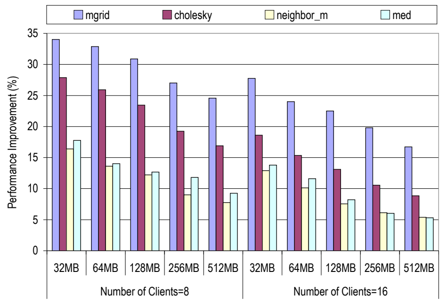


Fig. 16. Percentage performance improvements when the client-side cache capacity is changed (on the x-axis).

Figures 11, 12, and 13 together, we can conclude that our approach is robust under different number of I/O nodes and varying cache sizes.

Recall that the number of epochs used for both applications was 100. To evaluate the impact of a different epoch count, we conducted another set of experiments where the number of epochs is changed. Note that, the experiments are performed using a single I/O node with the default cache size, 256MB. The results are presented in Figure 14. We see that an epoch count of 100 generates the best result among all other counts tested in the experiments. When the numbers of epochs is very small (i.e., when the epochs are very large), we cannot capture the fine grain modulations in harmful prefetch patterns during execution. At the other extreme, when the number of epochs is very small, the overheads involved become too much. While we believe that it is possible to develop an enhanced scheme that adapts the epoch size to the runtime behavior of the application, we postpone this study to a future work.

As mentioned earlier, the default threshold values used in the coarse grain and fine grain versions of our schemes were 35% for both throttling and pinning. We also performed experiments with other threshold values, and the results are presented in Figure 15 for the coarse grain version (the trends with the finer grain version are very similar; so, we do not present them). Again, the results are obtained using a single I/O node with 256MB cache size. We see that, as expected, the percentage savings are significantly effected by the threshold value employed. A very low threshold value can invoke frequent throttles and pinnings, and this can in turn lead to poor performance. On the other hand, a very large threshold value can cut the number of (potentially useful) throttles and pinnings significantly, leading again to poor performance. As in the case of the number of epochs, it may be possible to develop a runtime strategy which can modulate the threshold value dynamically during the course of execution. However, detailed investigation of such a scheme is beyond the scope of this paper.

The default client-side cache capacity used in our experiments was 64MB. We also performed a sensitivity experiment to see how the size of client cache affects the performance savings our scheme achieves. The results for different sizes are presented in Figure 16 (the values of other parameters are as in their default configuration). We can see that, while the savings we achieve generally reduce as we increase the capacity of the client-side cache, the results are still good. In particular, even with the largest client cache we used, our fine grain scheme achieves about 14.6% average improvement in execution latency with 8 clients and 9.1% improvement with 16 clients.

Comparison with Simple Prefetching. We now present the results obtained when a simpler I/O prefetching algorithm is used. Recall that the prefetching approach used so far in our experimental evaluation is a compiler based one [25]. As explained earlier in Section II, this algorithm makes use of data reuse analysis to identify the data blocks for which to issue prefetches and most suitable points in the code to insert explicit prefetch instructions. As a result, it is careful in inserting prefetches and this helps minimize the number of unnecessary or useless prefetches. Consequently, this approach also help us cut the number of harmful I/O prefetches. To

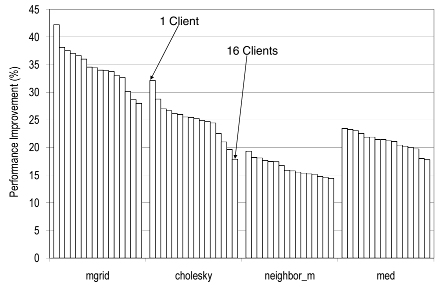


Fig. 17. Results obtained when a simpler I/O prefetching algorithm is used (fine grain version).

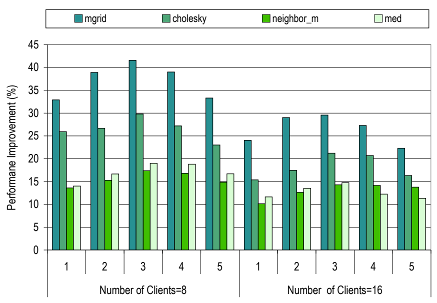


Fig. 18. Percentage savings when the value of the K parameter is changed (on the x-axis).

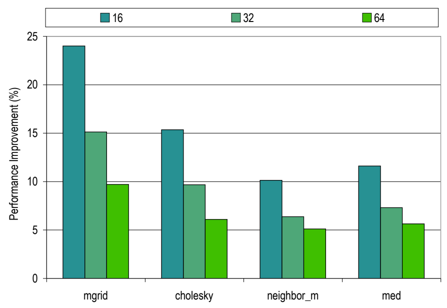


Fig. 19. Percentage savings when the number of clients is increased (on the x-axis).

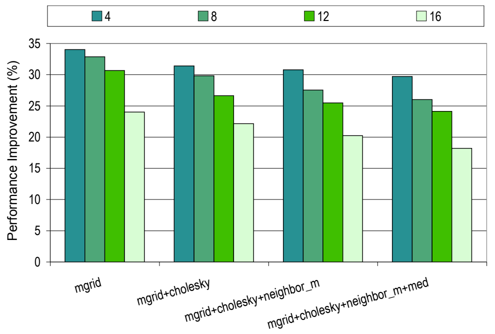


Fig. 20. Percentage improvements for mgrid when executed with multiple applications.

quantify the effectiveness of our approach under a different I/O prefetch scheme, we also implemented a simpler I/O prefetching scheme whereby whenever a data block is fetched (not through prefetching) from disk to memory cache, the next block on the same disk is prefetched automatically. Clearly, as compared to the compiler based scheme, this simpler scheme can issue many more prefetches (which means more harmful prefetches as well). The results with this scheme when used along with the fine grain version of our approach are shown in Figure 17 for the single I/O node case. When we compare these results to those given in Figure 10, it is easy to see that our approach generates better savings with the simple I/O prefetch scheme for all the client counts tested. The main reason for this is the fact that, as the number of prefetches increases, the percentage of harmful prefetches also increases. For example, although we do not present here in detail, for the 8 client case, when we move from the compiler based prefetching scheme to this simpler scheme, we observed that the percentage of harmful prefetches increased by 34.4%, 25.5%, 15.9%, and 21.4%, in applications mgrid, cholesky, neighbor_m, and med, respectively. Since prefetch throttling and data pinning targets harmful prefetches, their effectiveness increases with the simple scheme.

Extended Epochs. As explained earlier in detail, when our approach decides to enable, during epoch e, prefetch throttling and data pinning for epoch e+1, in the next epoch (e+2), throttling and data pinning are disabled for the involved clients. However, this can potentially lead to missing some optimization opportunities. In particular, epoch e+2 (and the next several epochs that follow it) could also benefit from the throttling and pinning decisions taken for epoch e+1. In other words, the same harmful prefetch pattern can last more than two consecutive epochs, and we could achieve additional performance improvements if we could take advantage of this. To test this, we performed another set of experiments where the throttling and pinning decisions taken during epoch e are applied to epochs e+1 through $e+K$ (inclusive), where K is a parameter that can be changed. Note that in our experiments discussed up to this point K is set to 1. For 8 and 16 client cases under the fine grain version of our approach, Figure 18 gives the percentage savings when the value of K is changed from 1 to 5. Our main observation from these results is that, as the value of K is increased, the percentage improvements first increase, and then beyond a point, they start to decrease. This means that a typical harmful prefetch pattern lasts 2-3 consecutive epochs but does not go beyond that (in general). Therefore, setting the value of K to 3 seems to be the right choice (we reached the same conclusion with the other client counts and our coarse grain version as well).

Scalability and Multiple Applications. Another aspect to explore is the scalability of our approach. To test how our savings look like when the number of clients is increased, we performed experiments with 32 and 64 clients. The results are presented in Figure 19; the results with 16 nodes are also reproduced for ease of comparison. We see that, while the savings are reduced as we increase the number of clients, we still achieve more than 5% improvement in all cases tested. The main reason for not achieving better saving figures is the (relatively) small data set sizes our applications use.

In our experiments so far, we assumed that an I/O node is not shared by multiple applications. When an I/O node is shared, our approach is still applicable as it is client-based, that is, it does not matter from an operational viewpoint whether the threads that are running on clients involved a harmful prefetch belong to the same application or different applications. However, in the case different applications, one can expect more irregularity in (Prefetching client, Affected client) plots shown earlier (see Figure 5). Figure 20 presents the percentage improvements for mgrid when its running alone, with one additional application, with two additional applications, and with three additional applications. We see that our approach performs very well under the multiple application scenario as well. The savings achieved are not as good as the case when no I/O node is shared. This is due to, as stated above, the fact that harmful prefetching patterns are more irregular when multiple applications are involved.

Comparison to Optimal Scheme. In this part of our experimental analysis, we compare our savings to those obtained by a hypothetical optimal scheme which assumes perfect

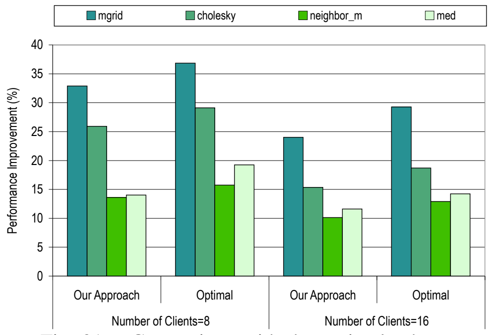


Fig. 21. Comparison with the optimal scheme.

knowledge about future data access patterns. The results with this scheme, given in Figure 21, are obtained using traces from our applications. This hypothetical scheme eliminates harmful prefetches in an optimal fashion. That is, for each prefetch, it determines whether it will be harmful or not, and if it will be harmful, that prefetch is dropped. For ease of comparison, Figure 21 also reproduces the results with our scheme (the fine grain version). We can see from these results that the difference between our scheme and the optimal one is not too much in these applications (the average difference being 3.6%).

VII. 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 [8], has been widely used to manage buffer caches, there are various approximations and enhancements to this, for example, the classical CLOCK algorithm [7]. While variants of the CLOCK algorithm are still dominant in many implementations, they also have some limitations in adaptability as far as changing access patterns is concerned. To mitigate this problem, several researchers studied enhancements to the classical CLOCK algorithm, such as 2Q [17] and LRFU [21]. More recent studies that try to handle accesses with weak temporal or spatial locality include CAR (Clock with Adaptive Replacement) [2], LIRS (Low Inter-reference Recency Set) [16], ARC (Adaptive Replacement Cache) [24], CLOCK-Pro [14], Second-Tier Cache Management [23], MultiQueue [38], and DULO (DUal LOcality) [15]. In some I/O-intensive applications where data access patterns are predictable, more efficient buffer cache management can be achieved by balancing caching against aggressive prefetching using the extracted future access patterns. Patterson et al [27] used a hint mechanism, which is designed to expose access patterns to the runtime system, in managing prefetching and caching file cache blocks.

I/O prefetching is also a very effective way of improving I/O performance. The basic idea is to hide I/O stall time by issuing I/O commands ahead of the time when they are actually needed. Since prefetched blocks can pollute the cache for normal cached blocks, prefetching should be designed and implemented carefully. Mowry et al [25] used compiler-generated information to manage prefetch commands more effectively in the context of single CPU execution. They also studied the cases where processes running concurrently on the same CPU generate I/O prefetch commands simultaneously [3]. To handle the interaction among multiple I/O prefetch instructions coming from multiple processes, they also introduced a *release* command in addition to compiler-inserted prefetch command [3]. Li and Shen [22] proposed a memory management framework that handles non-accessed but prefetched pages separately from the rest of the memory buffer cache. They employ different heuristic policies to select a victim page. They either evict the last page of the longest prefetch stream, or the last page of the least recently accessed prefetch stream, or evict the last page of the prefetch stream whose owner process has consumed the most amount of CPU since it last accessed the prefetch stream. For sequential workloads, Gill and Modha proposed a novel prefetching scheme, called SARC [11], that dynamically adapts the cache space between sequential and random streams. In TIP (Transparent Informed Prefetching) [27], "prefetch horizon" is used to limit the prefetch depth, beyond which there is no benefit from prefetching. While prefetch horizon can mitigate the impact of aggressive prefetching on effective cache size, this approach does not specifically target harmful prefetches. Patterson et al also studied the same problem under multi-process execution environments [31]. Kimbrel et al [28] studied the prefetching and caching in a system with parallel disks. More recently, Ding et al proposed a scheme, called DiskSeen [9], that improves I/O prefetching by exploiting disk layout and access patterns observed to overcome the inherent limitations of prefetching at the logical file level. Our approach described in this paper is different from these prior studies that considered I/O prefetching, since our main goal is to eliminate harmful prefetches that lead to intra- and inter-client misses on the shared storage caches. Gill and Bathen [10] also addressed a similar problem as ours does using a technique, called AMP, that determines the prefetching trigger point and the prefetching degree adaptively. Our approach is different from AMP because we evaluated the effect of harmful prefetches generated by applications running on multiple clients, whereas AMP's goal is to reduce harmful prefetches based on workload intensity and storage system load. Also, while AMP studied prefetching for commercial workloads, our focus is on compiler-directed I/O prefetching in parallel applications. Therefore, the approaches used in our work and [10] are entirely different from each other.

Targeting multi-level caches, several multi-level buffer cache management policies have been proposed [6], [36], [33], [37]. The main idea behind these approaches is to manage cache blocks in an exclusive manner, that is, a file cache block in one level of cache hierarchy does not present in another level of caches. To handle this, [36] introduced a DEMOTE operation where an evicted cache block is migrated to lower level of buffer cache. Chen et al [6] 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 [37] proposed an approach, called Karma, that uses application hints in managing the contents of a multi-level cache hierarchy.

VIII. Conclusions and Future Work

I/O prefetching is an important optimization that can potentially improve the performance of large scale data-intensive applications that perform disk I/O. The main contribution of this paper is to show that harmful prefetches can be very problematic for shared memory caches and to propose two complementary schemes that help reduce the negative impact of these prefetches. Prefetch throttling prevents one or more clients from issuing further prefetches if such prefetches are predicted to be harmful, i.e., discard the useful data accessed by other clients. Data pinning on the other hand makes selected data blocks immune to harmful prefetches by pinning them in the memory cache. This paper discusses both coarse grain and fine grain versions of these two optimization schemes and present a detailed experimental analysis. Our results demonstrate that the proposed prefetch throttling and data pinning schemes improve the performance of I/O prefetching significantly for the four application codes tested. Specifically, with the default values of our experimental parameters, our schemes reduce execution cycles by 15.1% over the no-prefetch case and 9.7% over the standard compiler-guided I/O prefetching, when 8 clients are used. The results also show that our savings are very close to those achieved by a hypothetical optimal scheme.

This work is a step toward adapting the behavior of I/O prefetching at runtime to improve performance. In our future work, we would like to extend our proposed system to more adaptive to different classes of applications, cache sizes, concurrently-executing multiple applications, etc.

Acknowledgment

We would like to thank our shepherd, Xiaosong Ma, for her useful feedback, and the anonymous reviewers for their insightful comments. This work is supported in part by NSF grants #0406340, #0444158, #0621402, #0724599, #0821527, and #0833126.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks, Summary and Preliminary Results. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 158–165, 1991.

[2] S. Bansal and D. S. Modha. CAR: Clock with Adaptive Replacement. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 187–200, 2004.

[3] A. D. Brown and T. C. Mowry. Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently. In *Proceedings of the Symposium on Operating System Design & Implementation*, pages 31–44, 2000.

[4] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the Annual Linux Showcase and Conference*, pages 317–327, 2000.

[5] C.-L. Chee, H. Lu, H. Tang, and C. V. Ramamoorthy. Improving I/O Response Times via Prefetching and Storage System Reorganization. In *Proceedings of the International Computer Software and Applications Conference*, pages 143–148, 1997.

[6] Z. Chen, Y. Zhou, and K. Li. Eviction-based Cache Placement for Storage Caches. In *Proceedings of the USENIX Annual Technical Conference*, pages 269–281, 2003.

[7] F. J. Corbato. A Paging Experiment with the Multics System, 1969.

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

[9] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *Proceedings of the USENIX Annual Technical Conference*, pages 261–274, 2007.

[10] B. S. Gill and L. A. D. Bathen. AMP: Adaptive Multi-stream Prefetching in a Shared Cache. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, pages 185–198, 2007.

[11] B. S. Gill and D. S. Modha. SARC: Sequential Prefetching in Adaptive Replacement Cache. In *Proceedings of the USENIX Annual Technical Conference*, pages 293–308, 2005.

[12] B. C. Gunter, W. C. Reiley, and R. A. V. D. Geijn. Parallel Out-of-core Cholesky and QR Factorizations with Pooclapack. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 1885–1894, 2001.

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

[14] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: an effective improvement of the CLOCK replacement. In *Proceedings of the USENIX Annual Technical Conference*, pages 35–35, 2005.

[15] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Localities. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2005.

[16] S. Jiang and X. Zhang. LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 31–42, 2002.

[17] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the International Conference on Very Large Data Bases*, pages 439–450, 1994.

[18] M. Kallahalla and P. J. Varman. Optimal Prefetching and Caching for Parallel I/O Sytems. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 219–228, 2001.

[19] M. S. Lam and M. E. Wolf. A Data Locality Optimizing Algorithm. *SIGPLAN Not.*, 39(4):442–459, 2004.

[20] D. T. Larose. *Data Mining Methods and Models*. John Wiley & Sons, 2006.

[21] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 134–143, 1999.

[22] C. Li and K. Shen. Managing Prefetch Memory for Data-Intensive Online Servers. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2005.

[23] X. Li, A. Aboulnaga, K. Salem, A. Sachedina, and S. Gao. Second-Tier Cache Management Using Write Hints. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2005.

[24] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 115–130, 2003.

[25] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 3–17, 1996.

[26] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. *SIGPLAN Not.*, 27(9):62–73, 1992.

[27] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 79–95, 1995.

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

[29] 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.

[30] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, 1999.

[31] A. Tomkins, R. H. Patterson, and G. Gibson. Informed Multi-Process Prefetching and Caching. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 100–114, 1997.

[32] 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.

[33] M. Vilayannur, A. Sivasubramaniam, M. T. Kandemir, R. Thakur, and R. B. Ross. Discretionary Caching for I/O on Clusters. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 96–103, 2003.

[34] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Not.*, 29(12):31–37, 1994.

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

[36] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the USENIX Annual Technical Conference*, pages 161–175, 2002.

[37] G. Yadgar, M. Factor, and A. Schuster. Karma: Know-it-All Replacement for a Multilevel Cache. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 25–25, 2007.

[38] Y. Zhou, J. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the USENIX Annual Technical Conference*, pages 91–104, 2001.