The following paper was originally published in the
Proceedings of the USENIX 2nd Symposium on
Operating Systems Design and Implementation
Seattle, Washington, October 1996

Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications

Todd C. Mowry, Angela K. Demke and Orran Krieger
University of Toronto

Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications

Todd C. Mowry, Angela K. Demke and Orran Krieger

Department of Electrical and Computer Engineering
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 3G4
{*tcm,demke,okrieg*}*@eecg.toronto.edu*

Abstract

Current operating systems offer poor performance when a numeric application's working set does not fit in main memory. As a result, programmers who wish to solve "out-of-core" problems efficiently are typically faced with the onerous task of rewriting an application to use explicit I/O operations (e.g., read/write). In this paper, we propose and evaluate a fully-automatic technique which liberates the programmer from this task, provides high performance, and requires only minimal changes to current operating systems. In our scheme, the compiler provides the crucial information on future access patterns without burdening the programmer, the operating system supports non-binding *prefetch* and *release* hints for managing I/O, and the operating system cooperates with a run-time layer to accelerate performance by adapting to dynamic behavior and minimizing prefetch overhead. This approach maintains the abstraction of unlimited virtual memory for the programmer, gives the compiler the flexibility to aggressively move prefetches back ahead of references, and gives the operating system the flexibility to arbitrate between the competing resource demands of multiple applications. We have implemented our scheme using the SUIF compiler and the Hurricane operating system. Our experimental results demonstrate that our fully-automatic scheme effectively hides the I/O latency in out-of-core versions of the entire NAS Parallel benchmark suite, thus resulting in speedups of roughly twofold for five of the eight applications, with two applications speeding up by threefold or more.

1 Introduction

Many of the important computational challenges facing scientists and engineers today involve solving problems with very large data sets. For example, global climate modeling, computational physics and chemistry, and many engineering problems (e.g., aircraft simulation) can easily involve data sets that are too large to fit in main memory [7, 9, 23]. For such applications (which are commonly referred to as "out-of-core" applications), main memory simply constitutes an intermediate stage in the memory hierarchy, and the bulk of the data must reside on disk or other secondary storage. Ideally one could efficiently solve an out-of-core problem by simply taking the original in-core program and increasing the problem size. In theory, a paged virtual memory system could provide this functionality by transparently migrating data between main memory and disk whenever page faults occur. While this approach does yield a logically correct answer, the resulting performance is typically so poor that it is not considered a viable technique for solving out-of-core problems [35].

In practice, scientific programmers who wish to solve out-of-core problems typically write a separate version of the program with explicit I/O calls for the sake of achieving reasonable performance. Writing an out-of-core version of a program is a formidable task—it is not simply a matter of inserting a few I/O read or write statements, but often involves significant restructuring of the code, and in some cases can have a negative impact on the numerical stability of the algorithm [35]. Thus the burden of writing a second version of the program (and ensuring that it behaves correctly) presents a significant barrier to solving large scientific problems.

1.1 The Problem with Paged Virtual Memory

The performance of out-of-core applications that rely simply on paged virtual memory to perform their I/O is typically quite poor, as we will see later in Section 4. In our experiments, the performance loss is *not* due to limited I/O bandwidth (in fact, the disk utilization is fairly low), but rather to I/O *latency*, since each page fault causes the application to suffer the full latency of fetching the data from disk.

One can potentially achieve better performance by rewriting out-of-core applications to use explicit I/O calls (e.g., the read/write interface in UNIX) for the following three reasons. First, the non-blocking I/O calls provided by asynchronous I/O interfaces allow an application to hide latency by overlapping disk I/O with computation. For example, if non-blocking disk reads can be scheduled early enough, then all of the latency can potentially be hidden. In contrast, a disk read under paged virtual memory typically does not begin until it is triggered by a page fault, at which point the application suffers the full latency of the disk I/O. Second, explicit I/O calls can fetch a large number of blocks in a single request, which is important to fully exploit the underlying parallelism in high-bandwidth I/O systems (e.g., disk arrays). In contrast, page faults typically result in only a single outstanding page-sized read request at a time for a given process. (Although most operating systems attempt some form of page fault prefetching both to hide latency and to have multiple outstanding disk requests, it is difficult to do this efficiently for reasons we will discuss later in Section 2.2.) Finally, the explicit management of memory buffer space under explicit I/O allows the application to minimize memory consumption by immediately freeing (and if dirty, writing out to disk) any buffers containing data that will not be accessed again in the near future. Under paged I/O, since the memory manager lacks application-specific knowledge of future access patterns, it may make bad decisions and free pages that are about to be accessed. To avoid this problem, the memory manager is typically conservative by allocating more memory to the application than is actually required, which may result in poor resource utilization.

1.2 The Problem with Explicit I/O

While explicit I/O offers the potential for improved performance over paging, it unfortunately suffers from several disadvantages. The primary disadvantage is the large burden placed on the programmer of rewriting an application to insert the I/O calls—our goal is to avoid this burden altogether. Another disadvantage is the performance overhead of these I/O system calls, which typically involve copying overhead to transfer data between the system's I/O buffers and the buffers managed by the application.

A third, less obvious disadvantage is that with explicit I/O, the application is implicitly making low-level policy decisions with its I/O requests (e.g., the size of the requests, and the amount of memory to be used for I/O buffering). However, the best policy decisions depend not only on application access patterns, but also on the physical resources available. Hence an application written assuming a particular amount of physical memory and disk bandwidth may perform poorly on a machine with a different set of resources, or in a multiprogrammed environment where some of the resources are being used by other applications. To illustrate how the available physical resources affect an application's performance, consider the amount of memory available for buffering I/O. If sufficient physical memory is available such that the entire data set can fit in memory, then an application with explicit I/O will pay the system call overhead with no benefit. On the other hand, if the application uses more buffer space for I/O than the available physical memory, then the buffers will suffer page faults, possibly resulting in worse performance than if the application had simply relied on paged virtual memory from the start.

1.3 Our Solution

To achieve high performance in out-of-core applications, we propose and evaluate a fully-automatic scheme for prefetching I/O whereby the operating system and the compiler cooperate to combine the advantages of both explicit I/O and paged virtual memory without suffering from the disadvantages. In our scheme, the compiler provides the crucial information on future access patterns without burdening the programmer, the operating system provides a simple interface for managing I/O which is optimized to the needs of the compiler, and a run-time layer accelerates performance by adapting to dynamic behavior and minimizing prefetch overhead. Our experimental results demonstrate that our scheme effectively hides the I/O latency in out-of-core versions of the entire NAS Parallel benchmark suite [2], thus resulting in speedups of roughly twofold for the majority of these applications, and over threefold in one case.

This paper is organized as follows. We begin in Section 2 by discussing how the compiler and the operating system can cooperate to automatically prefetch disk accesses for out-of-core applications. Next, in Sections 3 and 4, we describe our experimental framework and present our experimental results. Finally, in Sections 5, 6, and 7, we present related work, future work, and conclusions.

2 Automatically Tolerating I/O Latency

This section describes our system for automatically tolerating I/O latency. We begin by discussing the fundamental challenges that we have overcome, we then present an overview of our system, and finally we discuss the three major components of the system (i.e. the compiler, operating system, and run-time layer support) in more detail.

2.1 Fundamental Performance Issues

Our goal is to fully hide I/O latency, thus eliminating its impact on overall execution time. Conceptually, one can view our approach as enhancing the performance of virtual memory, since that is the abstraction we present to the programmer. Under paged virtual memory, an out-of-core application invokes two types of disk accesses: (i) faulting pages are read from disk into memory, and (ii) dirty pages are written out to disk to free up memory. Hiding write latency is reasonably straightforward since writes can be buffered and pipelined. Hiding *read* latency, on the other hand, is difficult because the application stalls waiting for the read (i.e. the page fault) to complete. The key to tolerating read latency is to split apart the *request* for data and the *use* of that data, while finding enough useful work to keep the application busy in between. We can accomplish this by *prefetching* pages sufficiently far in advance in the execution stream such that they reside in memory by the time they are needed.

Since prefetching does not reduce the number of disk accesses, but simply attempts to perform them over a shorter period of time, it cannot reduce the execution time of an application whose I/O bandwidth demands already outstrip the bandwidth provided by the hardware. Fortunately, we can construct cost-effective, high-bandwidth I/O systems by harnessing the aggregate bandwidth of multiple disks [5, 16, 28]. Roughly speaking, one can always increase the I/O bandwidth by purchasing additional disks.[1]

In addition to hiding I/O latency and providing sufficient I/O bandwidth, a third challenge in achieving high performance is effectively managing main memory, which can be viewed as a large, fully-associative cache of data that actually resides on disk. There are two issues here. First, to minimize page faults, we would like to choose the optimal page to evict from memory when we need to make room for new pages that are being faulted in. Toward this goal, most commercial operating systems use an approximation of LRU replacement to select victim pages. While LRU replacement may be a good choice for a default policy, there are cases where it performs quite poorly, and in such cases we would like to exploit application-specific knowledge to choose victim pages more effectively. The second issue is that we would like to minimize memory consumption, particularly when doing so does not degrade performance. For example, rather than filling up all of main memory with data that we are streaming through, we may be able to achieve the same performance by using only a small amount of memory as buffer space. By minimizing memory consumption, more physical memory will be available to the rest of the system, which is particularly important in a multiprogrammed environment. To accomplish both of these goals, we introduce an explicit *release* operation whereby the application provides a hint to the OS that a given page is not likely to be referenced again soon, and hence is a good candidate for replacement.

In summary, our approach overcomes the fundamental challenges of accelerating paged virtual memory as follows: (i) *prefetches* are used to tolerate disk read latency, (ii) *multiple disks* are used to provide high-bandwidth I/O, and (iii) *release* operations are used to effectively manage memory. We now discuss the overall structure of our software system.

2.2 Software Architecture Overview

To prefetch and release data effectively, we need detailed knowledge of an application's future access patterns. Although one might attempt to deduce this information from inside the OS by looking for repeated patterns in the access history, such an approach would be limited only to simple access patterns (e.g., even the simple indirect refer-

[1] There are more subtle issues involved with increasing I/O bandwidth, of course. However, this approach does appear to be promising enough for our purposes, and exploring I/O bandwidth issues further is beyond the scope of this paper.

ences that commonly occur in sparse-matrix applications would be extremely difficult for the OS to predict), and would require adding additional complexity to the OS, which is something we wish to avoid.[2] Instead, we turn to the compiler to provide information on future access patterns, since it has the luxury of being able to examine the entire program all at once. Also, by using the compiler to extract this information automatically, we avoid placing any burden on the programmer, who continues to enjoy the abstraction of unlimited virtual memory.

2.2.1 The Compiler / OS Interface

Given that the compiler will be extracting and passing access pattern information to the OS, an important question is what form this interface should take. Note that this interface will *only* be used by the compiler, and *not* by the programmer—the programmer's interface will be unlimited virtual memory, and the compiler and operating system cooperate to preserve this illusion. Ideally, we would like an interface that requires minimal complexity within the OS (so that it can be readily incorporated into an existing commercial OS), and which maximizes the compiler's ability to improve performance, given the strengths and weaknesses of realistic compilation technology.

One possibility would be for the compiler to pass a summary of future access patterns to the OS through a single call at the start of execution. However, from the compiler's perspective, this approach is undesirable since the access patterns in real applications often depend on dynamic control and data dependencies that can only be resolved at run-time. For example, in the bucket sort application (BUK) discussed later in this paper, the important data accesses are indirect references based on the contents of a large array. The values in this array are unknown at startup time; but even if they were known, passing this very large array along with a description of how to use it to compute addresses would greatly complicate not only the interface and the compiler, but also the OS, which would ultimately be responsible for generating the addresses. Another disadvantage of this approach is that it pushes the complexity of matching up the access patterns with *when* those accesses actually take place into the OS. For example, if the compiler indicates that the program will be streaming through a large array, it is not helpful if the OS brings the data into memory too fast (or too slow) relative to the rate at which it is being consumed. Since tracking an application's access patterns means that the OS must see either page faults or explicit I/O on a regular basis, it is unclear that this interface offers any less overhead than an interface requiring regular system calls. Hence we will focus instead on an interface where prefetch addresses are passed in at roughly the time when the prefetch should be sent to disk, and where release addresses are passed in when the data is no longer needed.

The next logical question is whether we can simply compile to an existing asynchronous read/write I/O interface, or whether a new interface is actually needed. There are two reasons why existing read/write I/O interfaces are unacceptable for our purposes. First, for the compiler to successfully move prefetches back far enough to hide the large latency of I/O, it is essential that prefetches be *non-binding* [19]. The non-binding property means that when a given reference is prefetched, the data value seen by that reference is bound at *reference* time; in contrast, with a binding prefetch, the value is bound at *prefetch* time. The problem with a binding prefetch is that if another store to the same location occurs during the interval between a prefetch and a corresponding load, the value seen by the load will be stale. Hence we cannot move a binding prefetch back beyond a store unless we are certain that they are to different addresses—unfortunately, this is one of the most difficult problems for the compiler to resolve in practice (i.e. the problem of "alias analysis", also known as "memory disambiguation" or "dependence analysis"). Since an asynchronous I/O read call implicitly renames data by copying it into a buffer, it is a binding prefetch. To illustrate this problem, consider the code in Figure 1(a). If we use the read/write interface, we might generate code similar to Figure 1(b). Unfortunately, this code produces an incorrect result if the parameters `a` and `b` are aliased (e.g., `foo(&X[0],&X[0])`) or even partially overlap (e.g, `foo(&X[10],&X[0])`). To implement non-binding prefetching, the data should have the same name (or address) both in memory and on disk, which corresponds to the abstraction of paged virtual memory. Figure 1(c) shows the preferred code which uses non-binding prefetch and release operations, and always produces a correct result.

The second problem with an asynchronous read/write interface is that it compels the OS to perform an I/O access. Instead, we would prefer to give the OS the flexibility to drop requests if doing so might achieve better performance, given

[2] This additional complexity may increase the critical page fault path in the OS, and hence degrade application performance.

```
foo(double *a, double *b) {
   /* Assume that a & b reside */
   /* on disk at this point. */
   ...
   for (i = 0; i < 100; i++) {
      a[i+1] = a[i] + b[i];
   }
}
```

(a) Original Code

```
foo(double *a, double *b) {
   double a_buf[101], b_buf[100];
   /* Read a & b from disk into buffers. */
   read(a,&a_buf[0],101*sizeof(double));
   read(b,&b_buf[0],100*sizeof(double));
   ...
   for (i = 0; i < 100; i++) {
      a_buf[i+1] = a_buf[i] + b_buf[i];
   }
   /* Write a_buf back out to disk. */
   write(a,&a_buf[0],101*sizeof(double));
}
```

(b) Read/Write Interface

```
foo(double *a, double *b) {
   /* Prefetch a & b into memory. */
   prefetch(a,101*sizeof(double));
   prefetch(b,100*sizeof(double));
   ...
   for (i = 0; i < 100; i++) {
      a[i+1] = a[i] + b[i];
   }
   /* Finished with a & b. */
   release(a,101*sizeof(double));
   release(b,100*sizeof(double));
}
```

(c) Prefetch/Release Interface

Figure 1: Example illustrating the importance of non-binding prefetches.

the dynamic demands for and availability of physical resources. For example, if there is not enough physical memory to buffer prefetched data, or if the disk subsystem is overloaded, we may want to drop prefetches. Hence the preferred interface is a natural extension of paged virtual memory which includes *prefetch* and *release* as non-binding performance hints, thus giving the compiler the flexibility to aggressively move prefetches back ahead of references, and giving the OS the flexibility to arbitrate between the competing resource demands of multiple applications. (Note that the "`MADV_WILLNEED`" and "`MADV_DONTNEED`" hints to the `madvise()` interface can potentially be used to implement prefetch and release in UNIX.)

2.2.2 Minimizing Prefetch Overhead

Earlier studies on compiler-based prefetching to hide cache-to-memory latency have demonstrated the importance of avoiding the overhead of unnecessarily prefetching data that already resides in the cache [19, 20]. To address this problem, compiler algorithms have been developed for inserting prefetches only for those references that are likely to suffer misses. An analogous situation exists with I/O prefetching, since we do not want to prefetch data that already resides in main memory—hence, we perform similar analysis in our compiler (as we discuss later in Section 2.3). Unfortunately, it is considerably more difficult to avoid unnecessary prefetches with I/O prefetching since main memory is so much larger than a cache that our loop-level compiler analysis tends to underestimate its ability to retain data. As a result, unnecessary prefetches do occur, and we must be careful to minimize their overhead.

Compared with cache-to-memory prefetching, where the overhead of an unnecessary prefetch is simply a wasted instruction or two[3], the overhead of an unnecessary I/O prefetch is considerably larger since it involves making a system call and checking the page table before discovering that the prefetch can be dropped. To reduce this overhead, we introduce a run-time layer layer in our system which keeps track at the *user level* of whether pages are believed to be in memory or not. Therefore we can typically drop unnecessary prefetches immediately without performing a system call, and we have found this to be essential in achieving high performance.

Having introduced the three layers of our system—the compiler, the OS, and the run-time layer—we now discuss each layer in more detail.

2.3 Compiler Support

The bulk of our compiler algorithm is a straightforward extension of an algorithm that was developed earlier for prefetching cache-to-memory misses in dense-matrix and sparse-matrix codes [19, 20]. Roughly speaking, we changed the input parameters that describe the cache size, line size, and miss latency to correspond to main memory size, the page size, and the page fault latency, respectively. Based on this memory model, the compiler uses *locality analysis* to predict when misses (i.e. page faults) are likely to occur, it isolates these faulting instances through *loop splitting* techniques, and schedules prefetches early enough using *software pipelining*. Figure 2 shows an example of the output of our compiler for a simple loop body (notice that it is able to prefetch the indirect `a[b[i]]` reference as well as the dense `b[i]` and `c[i][j]`

[3] Unnecessary cache prefetches are dropped as soon as the primary cache tags are checked. The overhead is simply the prefetch instruction, plus any instructions needed to generate the prefetch address, plus one cycle of wasted cache tag bandwidth.

```
int a[1000000];
int b[1000000];
int c[1000000][8];

for (i = 0; i < 1000000; i++)
  for (j = 0; j < 8; j++)
    a[b[i]] = a[b[i]] + c[i][j];
```

(a) Original Code

```
prefetch_block(&b[0], 8);
prefetch_block(&c[0][0], 4);
for (i = 0; i < 128; i++)
  prefetch(&a[b[i]]);
/* Note: 995328 = (⌊1000000/4096⌋ - 1)*4096 */
for (i1 = 0; i1 < 995328; i1 += 4096) {
  prefetch_release_block(&b[i1+8192], &b[i1-1], 4);
  for (i0 = i1; i0 < i1 + 4096; i0 += 512) {
    prefetch_release_block(&c[i0+512][0], &c[i0-1][0], 4);
    for (i = i0; i < i0 + 512; i++) {
      prefetch(&a[b[128+i]]);
      for (j = 0; j < 8; j++)
        a[b[i]] = a[b[i]] + c[i][j];
    }
  }
}
for (i = 995328; i < 1000000; i++)
  for (j = 0; j < 8; j++)
    a[b[i]] = a[b[i]] + c[i][j];
```

(b) Code with Prefetching

Figure 2: Example of the output of the prefetching compiler. (The first argument to all prefetch calls is the prefetch address; the second argument to `prefetch_release_block` is the release address; the final argument to "block" versions is the number of 4KB pages to be fetched and/or released.)

references). Since space limitations prevent us from describing the compiler algorithm in detail, we focus mainly on the major changes to the original algorithm [19].

Two of our modifications to support I/O prefetching are related to spatial locality—i.e. when strided accesses fall within the same page—in which case page faults only occur on iterations that cross page boundaries. First, we use *strip mining* [24] rather than *loop unrolling* to isolate these faulting iterations, since replicating a loop body 1000 times or more is clearly infeasible. Notice in Figure 2(b) that loop `i` has been strip mined twice (into loops `i0` and `i1`) to account for the spatial locality of `b[i]` and `c[i][j]`. (The `i` loop has been strip mined twice since `c[i][j]` accesses data more quickly than `b[i]`, and therefore needs to be prefetched at a faster rate.) Second, to fully exploit the available bandwidth in our I/O subsystem, we prefetch several pages at a time for references with spatial locality (e.g., four pages are fetched at a time for `b[i]` and `c[i][j]`[4]). (Note that for references without spatial locality—e.g., `a[b[i]]`—we prefetch only a single page at a time.) Similarly, we convert the prolog loops from the original algorithm into block prefetches whenever possible, as shown in the first two lines of Figure 2(b).

Generating release operations is straightforward. The compiler already identifies groups of references that effectively share the same data and can be treated as a single reference—this is called "group locality". For each of these groups (a group may potentially contain only a single reference), the compiler identifies the *leading reference* (i.e. the first reference to access the data) as the reference to prefetch—we simply extend this analysis to also identify the *trailing reference* (the last one to touch the data) as the address to release. (Note that for indirect references such as `a[b[i]]`, we do not generate a release operation since it is too difficult to predict whether the data will be accessed again soon.) To minimize system call overhead, we bundle prefetch and release requests together whenever appropriate, as illustrated in Figure 2(b).

Perhaps the most significant change we made to the original algorithm is to reason more carefully about loop bounds and array bounds that are small relative to a page size. This was less of a concern for cache-to-memory prefetching due to the relatively small size of cache lines. However, it is common to find inner loops (and sometimes even surrounding loops) which access less data than a 4 KB page (e.g., the `j` loop in Figure 2(a)). Attempting to software pipeline prefetches across such loops is ineffective, since the pipeline never gets started. Instead, our compiler pipelines the prefetches across the first surrounding loop which touches more than a page of the given array, as illustrated by the fact that prefetches for `c[i][j]` are pipelined along the

[4] The number of pages to fetch in a block is a parameter which can be specified to the compiler. We chose four arbitrarily for this example.

`i` loop rather than the `j` loop in Figure 2(b). Having described the compiler support for I/O prefetching, we now focus on the other half of the equation: the operating system.

2.4 Operating System and Run-Time Layer Interaction

To support compiler directed prefetching, the OS needs to be able to respond to the prefetch and release operations issued by the application. This functionality is easy to add since the OS is already able to unmap pages of memory and initiate asynchronous requests to the file system. One issue is how the OS should handle prefetch requests when there is no free memory available. Since we expect the application/compiler to be managing memory requirements, the OS simply drops prefetches when all memory is in use.

To help the run-time layer reduce the overhead of prefetching, the OS also provides applications with a single physical memory page that is shared with the OS. Applications that prefetch are required to register with the OS to initiate sharing. The shared page is used as a bit vector with each bit representing one or more contiguous pages of the application's virtual memory space (a set bit indicates that the corresponding page is in memory). The granularity of the bit vector is determined by the run-time layer at program start-up. Bits are set by the run-time layer when a prefetch request is issued, and by the OS when non-prefetched page faults occur. The OS also clears bits when release requests are issued and when the memory manager reclaims pages.

The run-time layer uses the bit vector to *filter* the prefetches inserted by the compiler by checking to see if the requested page is already in memory. In many cases this simple test can avoid the cost of a system call to the OS, thus reducing overhead. For block prefetch requests, we check each page until one is found that is not in memory, then pass all remaining pages to the OS. In this way, at most one system call is required for a block prefetch.

3 Experimental Framework

We now describe our experimental platform, and the applications which we study in our experiments.

3.1 Experimental Platform

The experimental platform used to evaluate our scheme is the Hurricane file system [16] and Hurricane operating system [33] running on the Hector shared-memory multiprocessor [34]. Hurricane is a hierarchically clustered, micro-kernel based operating system that is mostly POSIX compliant. It was largely irrelevant that the system was a multiprocessor; we chose this platform because the system has multiple disks attached to it, the file system can stripe a single file across multiple disks, and the operating system could be modified to add prefetch and release operations. For all experiments shown in subsequent sections, the pages of the applications are striped by the file system round-robin across all seven disks. An extent-based policy is used to store the file on each of the disks, where contiguous file blocks are stored to contiguous blocks on the disk to avoid seek operations for sequential file accesses. The disk scheduler treats prefetches the same as normal disk read requests.

Table 1: Experimental platform characteristics.

Processor	
Processor type:	Motorola 88100
Clock rate:	16.67 MHz
Data cache size:	16KB
Instruction cache size:	16KB
Physical Memory	
Total size:	64 MBytes
Available to application:	48 MB
Disks	
Number of disks:	7
Maximum transfer rate:	640 KB/sec
Average rotational latency:	8.61 msec
Track-to-track seek time:	5 msec
Kernel Operation Overhead	
IPC request:	70 μsec
In-core fault:	200 μsec
Out-of-core fault:	800 μsec
Base prefetch:	60 μsec
+ per out-of-core page:	200 μsec
+ per in-core page:	30 μsec
+ per in-page table page:	10 μsec
File System Operation Overhead	
Prefetch (per-page):	70 μsec
Read/Write (per-page):	70 μsec

In addition to adding prefetch and release operations to Hurricane, we also added extensive instrumentation to enable us to produce the detailed statistics shown in subsequent sections. The basic characteristics of our experimental platform (with the instrumentation disabled) are shown in Table 1, and more detailed descriptions of the platform can be found in earlier publications [16, 33, 34].

We believe that our experimental results are conservative for the following reasons: (i) instrumentation is enabled for all the experiments, and

Table 2: Description of applications.

Name	Description	Input Data Set	Memory Required Absolute	Memory Required % of Available	Original Execution Time (mins)
BUK	integer bucket sort algorithm	2^{23} 19-bit integers	103 MB	215 %	21.0
CGM	solves an unstructured sparse linear system using the conjugate gradient method	28000x28000 sparse matrix with 7,607,024 non-zeros	103 MB	215%	57.2
EMBAR	monte-carlo simulation	2^{24} random numbers	134 MB	279%	53.9
FFTPDE	3-D FFT PDE, performs forward and inverse FFT's	128x128x128 matrix of complex numbers	117 MB	244%	87.9
MGRID	computes 3-D scalar potential field on a uniform cubical grid using a multigrid solver	128x128x128 matrix	58 MB	121%	31.9
APPLU	solves four coupled parabolic / elliptic PDE's using SSOR method to invert jacobian matrix	5x5x64x64x32 matrices	120 MB	250%	48.9
APPSP	solves five coupled parabolic / elliptic PDE's using diagonalized approximate factorization method	90x90x90 matrices	117 MB	244%	224.3
APPBT	solves three coupled parabolic / elliptic PDE's using block approximate factorization method	5x5x64x64x32 matrices	94 MB	196%	85.2

hence the system overheads are significantly inflated; (ii) the operating system overhead is also inflated because the hardware does not support cache coherence, and hence many of the operating system data structures are accessed in an uncached state; and (iii) processor speeds have increased more rapidly than disk speeds, and hence the importance of tolerating I/O latency has increased in modern systems.

3.2 Applications

To evaluate the effectiveness of our approach, we measured its impact on the performance of the entire NAS Parallel benchmark suite [2]. We chose these applications because they represent a variety of different scientific workloads, their data sets can easily be scaled up to out-of-core sizes, and they have not been written to manage I/O explicitly. Our goal is to show that these scientific benchmarks can achieve high performance with out-of-core data sets without requiring any extra effort to rewrite the program. Because these programs were originally written to evaluate processor performance, they all generate a data set at start-up, perform a series of computations, and then discard the results. To make the programs more realistic, we modified them to use a pre-initialized data set and write their results back out to disk. This was achieved by using a mapped file I/O interface—the data accesses have not been modified but the data now comes from disk. An exception to this strategy is EMBAR where a random initialization is performed once for every iteration and separation would not be appropriate. A brief description of each of the benchmarks and the data set used is given in Table 2.

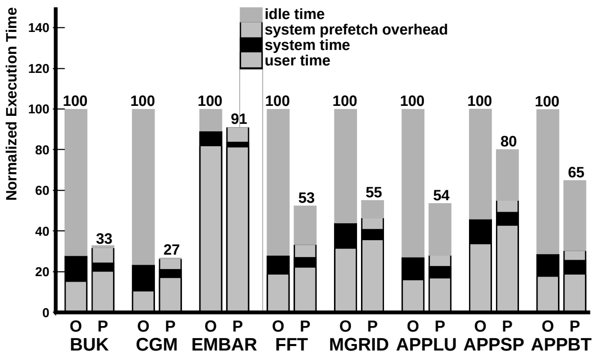
We implemented our prefetching algorithm as a pass in the SUIF (Stanford University Intermediate Format) compiler [31], which we used to convert the original Fortran source code of each application into C code containing prefetch and release calls (as illustrated earlier in Figure 2(b)). We then compile this resulting C code into a Hurricane executable using gcc version 2.5.8 with the -O2 optimization flag.

4 Experimental Results

We now present the results of our experiments. We begin by focusing on the impact of our scheme on overall execution time, including the effectiveness of the compiler and the run-time layer. We then look at the performance from a system-level perspective, including the effects on disk and memory utilization.

4.1 Performance Improvement

Figure 3(a) shows the overall performance improvement achieved through our automatic prefetching scheme. For each application, we show two bars representing normalized execution time: the original program relying simply on paged virtual memory to perform its I/O (**O**), and the program once it is compiled to use prefetching (**P**). In each bar, the top section is the amount of time when the proces-



(a) Overall Performance
(**O** = original, **P** = with prefetch)

Benchmark	Original		With Prefetch		
	Total Faults (x1000)	Avg. Stall Time (msec)	Total Faults (x1000)	Avg. Stall Time (msec)	I/O Stall Reduction (%)
BUK	41.529	24.5	0.810	16.1	98.7%
CGM	135.066	22.0	0.207	26.5	99.8%
EMBAR	65.535	7.7	0.005	13.5	100.0%
FFTPDE	135.646	31.1	28.432	39.3	73.6%
MGRID	62.231	19.9	7.642	24.2	85.1%
APPLU	91.220	26.3	31.663	26.4	65.2%
APPSP	412.234	20.5	143.996	26.2	55.4%
APPBT	156.172	26.2	77.035	25.6	51.9%

(b) I/O Stall Statistics

Figure 3: Overall performance improvement from prefetching

sor was idle, which corresponds roughly to the I/O stall time since we run only a single application during these experiments. The bottom section of each bar is the time spent executing in user mode—for the prefetching experiments, this includes the instruction overhead of issuing prefetches, including any overhead in the run-time layer of checking the bit vector to filter out unnecessary prefetches. The middle sections of each bar are the time spent executing in system mode. For the original programs, this is the time required for the operating system to handle page faults; for the prefetching programs, we also distinguish the time spent in the OS performing prefetch operations.

As we see in Figure 3(a), the speedup in overall performance ranges from 9% to 270%, with the majority of applications speeding up by more than 80%. Figure 3(b) presents additional information on page faults[5] and stall time. As we see in Figure 3(b), more than half of the I/O stall time has been eliminated in seven of the eight applications, with three applications eliminating over 98% of their I/O stall time.

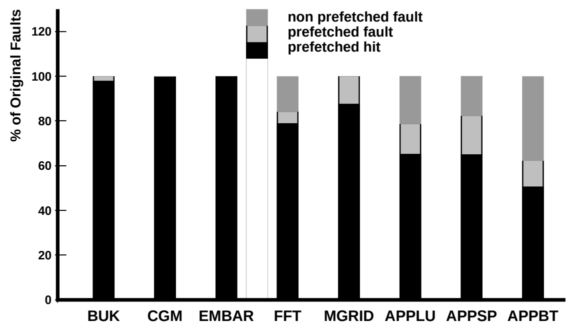
Having established the benefits of our scheme, we now focus on the costs. Figure 3(a) shows that the instruction overhead of generating prefetch addresses and checking whether they are necessary in the run-time layer causes less than a 20% increase in user time in five of the eight applications—in the worst case (CGM), the user time increases by 70%. However, in all cases this increase is quite small relative to the reduction in I/O stall time. If we focus on the system-level overhead of performing prefetch operations, we see in Figure 3(a) that in most cases this overhead is directly offset by a reduction in system-level overhead for processing page faults. Hence the overheads of our scheme are low enough to translate into significant overall performance improvements in all of these applications.

We wish to emphasize that all of these results are fully automatic—we have not rewritten any of the applications or modified the code generated by the compiler. Having discussed the performance at a high level, we now focus on the compiler and run-time layer in more detail.

4.1.1 Effectiveness of the Compiler and Run-Time Layer

Figure 4 presents additional information which is useful for evaluating how effective our compiler is at inserting prefetches appropriately, and how effective the run-time layer is at minimizing prefetching overhead. Figure 4(a) shows a breakdown of the impact of prefetching on the original page faults in the application. This breakdown contains three categories: (i) those that were prefetched and and successfully eliminated page faults (*prefetched hit*), (ii) those that were prefetched but remained page faults (*prefetched fault*), and (iii) those that were not prefetched (*non-prefetched fault*). The combination of the first two cases is often referred to as the *coverage factor* (i.e. the fraction of original page faults that were prefetched). For all cases except APPBT, the coverage factor is greater than 75% (in four cases, it is greater than 99%). Most of the page faults that we failed to prefetch were due to inner loops with small loop bounds, where the fact that the bound was small could not be determined at compile time. For example, if the `j` loop in example in Figure 2(a) had an upper bound of `N` which turned out to be small at run-time (but which we could not determine at compile time), and if the dimensions of the `c` matrix were also unknown at compile time, our compiler can make the mistake of software pipelining references across the

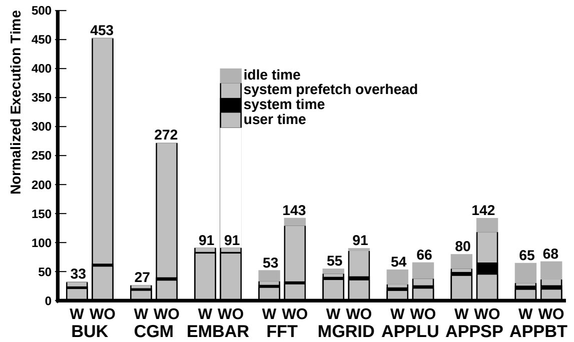
[5]Throughout this discussion, we will refer to page faults that cause the application to stall waiting for I/O simply as faults, and ignore page faults for in-core data.



(a) Coverage Factor

Benchmark	Unnecessary Prefetches Issued to OS	Inserted Prefetches Filtered at Run-Time
BUK	0.07%	99.79%
CGM	0.08%	99.74%
EMBAR	0.00%	0.02%
FFTPDE	7.99%	99.59%
MGRID	8.03%	99.17%
APPLU	3.75%	96.99%
APPSP	7.55%	99.51%
APPBT	2.54%	98.31%

(b) Unnecessary Prefetches



(c) Performance of prefetching with (**W**) and without (**WO**) filtering (normalized to the original, non-prefetched case).

Figure 4: Effectiveness of the compiler analysis and run-time filtering.

`j` loop rather than the `i` loop. Situations like this can cause us to miss important prefetches, since the software pipeline never gets started. This problem can be fixed through a straightforward extension of our compiler algorithm whereby we create two versions of the loop, and choose the proper one to execute by testing the loop bound at run-time.

The effectiveness of our compiler in scheduling prefetches the right amount of time in advance is reflected by the size of the *prefetched fault* category in Figure 4(a). A large value means that the prefetches are either not issued early enough, in which case the page has not arrived in memory by the time it is referenced, or are issued too early, in which case the page has already been flushed from memory before it is referenced. In the cases where this category is noticeable in Figure 4(a), the problem is almost always that the prefetches were not issued early enough. However, given how large I/O latency is, it is encouraging that this case is generally small relative to the number of successful prefetches.

To evaluate the effectiveness of the run-time layer at reducing prefetching overhead, Figure 4(b) presents statistics on how many prefetches were *unnecessary* (i.e. the page was already mapped into memory). (Note that a prefetch for a page that is in memory but is on the free list is not considered to be unnecessary, since it performs useful work by reclaiming the page.) The left-hand column of Figure 4(b) shows that almost all of the prefetches issued to the system by the run-time layer are useful. All unnecessary prefetches that are issued to the system occur as part of a block prefetch request in which prefetching is required for at least one page. The right-hand column of Figure 4(b) shows the fraction of dynamic prefetches that were inserted by the compiler which turn out to be unnecessary, and are filtered out by the run-time layer. For reasons discussed earlier in Section 2.2.2, it is difficult for our compiler to avoid inserting unnecessary prefetches, and we see that over 96% of the prefetches were unnecessary for all but EMBAR (where the access patterns are simple enough that the compiler's analysis is perfect).

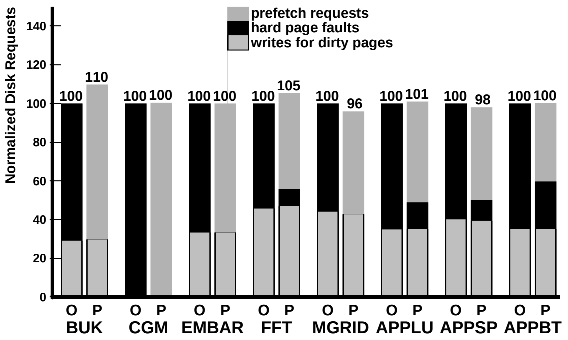
Figure 4(c) quantifies the performance advantage of the run-time layer. As we see in Figure 4(c), half of the applications (BUK, CGM, FFT and APPSP) run slower than the original non-prefetching versions when the run-time layer is removed. This is not surprising since the overhead of dropping an unnecessary prefetch in the run-time layer is roughly 1% as expensive as issuing it to the OS. Hence the run-time layer is clearly essential.

4.2 Disk and Memory Utilization

In Figure 5 we break down the types of requests seen by the disks and show average disk utilization during execution for both the original and prefetching versions of the applications. In almost all cases, the total disk requests do not increase as a result of prefetching, and for two of the applications they actually decrease as prefetches prevent the system

Table 3: Memory sub-system activity and amount of free memory

Benchmark	Original			With Prefetch and Release			
	Pages Freed by System (pages)	Minimum Free Memory (%)	Average Free Memory (%)	Pages Freed by System (pages)	Pages Freed by release (pages)	Minimum Free Memory (%)	Average Free Memory (%)
BUK	68916	5.8%	26.9%	3461	41729	29.2%	73.7%
CGM	125817	14.8%	21.1%	125710	834	7.3%	23.4%
EMBAR	55647	15.0%	22.0%	0	65504	98.4%	98.5%
FFTPDE	146699	14.9%	20.9%	156463	7164	9.9%	26.0%
MGRID	59181	14.3%	23.4%	60349	0	12.3%	25.9%
APPLU	82978	11.9%	25.0%	84395	0	7.9%	28.9%
APPSP	450507	10.5%	18.6%	448732	17196	9.0%	35.4%
APPBT	148174	11.3%	22.8%	148580	516	11.7%	25.5%



(a) Disk activity

Benchmark	Original	With Prefetch
BUK	11.8%	40.1%
CGM	11.6%	46.0%
EMBAR	5.9%	9.0%
FFTPDE	18.9%	35.1%
MGRID	15.8%	29.0%
APPLU	18.6%	31.8%
APPSP	16.3%	20.7%
APPBT	15.8%	20.1%

(b) Average disk utilization

Figure 5: Breakdown of requests sent to disk and average utilization (**O** = original program, **P** = with prefetch)

from writing out dirty pages that will be referenced again soon. Hence the increased disk utilization shown in Figure 5(b) is simply due to the fact that we are performing roughly the same number of disk accesses over a shorter period of time.

Finally, Table 3 summarizes memory usage during each application's execution. Since our current compiler implementation is not aggressive about inserting release operations, most applications do not contain a significant number of them. However, when release operations are used (e.g., BUK and EMBAR), we see that a large percentage of memory is kept free at all times since only the portion of the data set actually being used is kept in memory. We expect that this would greatly reduce the impact of an out-of-core program on other applications in a multiprogrammed environment, and we intend to explore this issue further in future work.

4.3 Problem Size Variations

Having demonstrated the benefits of I/O prefetching where the problem size is roughly twice as large as the available memory, we now look at the performance when the problem size is varied.

4.3.1 In-Core Problem Sizes

We begin with cases where the data sets fit within main memory. In these cases, we would expect prefetching to degrade performance, since the prefetches incur overhead but provide little or no benefit. Figure 6 shows two sets of experiments—the cold-started and warm-started cases—on data sets that are roughly 10-35% as large as the available memory. Starting with the cold-started cases, we see that prefetching degrades performance in four cases, but actually *improves* performance in three cases (BUK, APPLU, and APPBT) by hiding the latency of cold page faults. To further isolate the prefetching overhead, we also warm-started the applications by preloading all of their data from the input files into memory before timing the runs. As expected, prefetching typically degrades performance in the warm-started cases since it offers no potential advantage. However, we believe that the cold-started cases are more realistic for most applications, since real programs must read their input data from disk.

In these experiments, we made no attempt to minimize prefetching overhead for in-core data sets, but this is a problem that we are planning to ad-

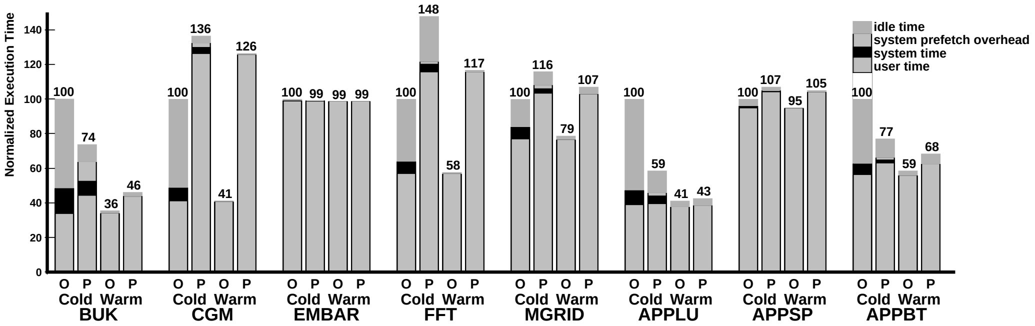


Figure 6: Performance with in-core data sets (**O** = original, **P** = with prefetch; **Cold** = cold-started, **Warm** = warm-started). Performance is normalized to the original, cold-started cases.

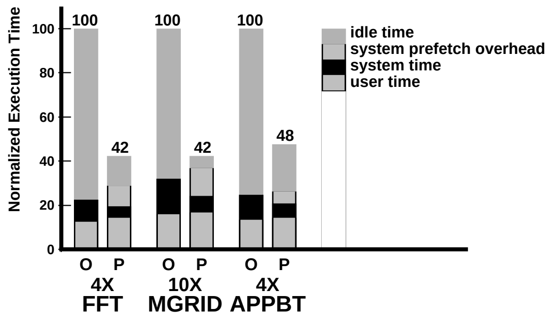


Figure 7: Performance with larger out-of-core problem sizes. Numbers above application names indicate how much larger the problem sizes are than available memory.

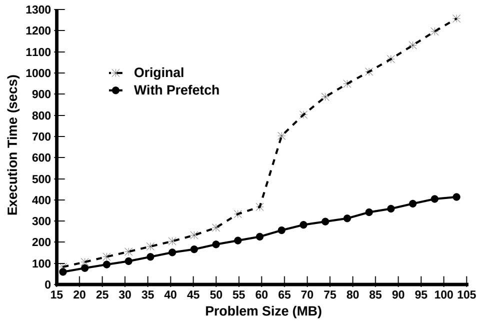


Figure 8: Performance of BUK (cold-started) across a range of problem sizes.

dress in future work. In particular, we can generate code that dynamically adapts its behavior by comparing its problem size with the available memory at run-time, and suppressing prefetches (after the cold faults have been prefetched in) if the data fits within memory. The fact that I/O prefetching can still potentially improve performance even on relatively small data sets by hiding cold page faults is an encouraging result.

4.3.2 Larger Out-of-Core Problem Sizes

In addition to looking at smaller problem sizes, we also experimented with much larger data sets than our earlier out-of-core problem sizes. Figure 7 shows the performance of three applications where the problem size is 4-10 times larger than the available memory. Recall that for MGRID, our earlier problem size was only 20% larger than the available memory–the next larger problem size (shown in Figure 7) requires 464 MB of memory, which is approximately 10 times more than what is available. In all three cases, the performance improvements remain large. In fact, prefetching offers slightly larger speedup in all three cases since there is more I/O latency to hide.

4.3.3 Case Study: BUK

To illustrate the impact of I/O prefetching on performance across a wide range of problem sizes, we look at the BUK application as a case study. We chose BUK because we can easily set the problem size to any value for this application. Ignoring page faults, we would normally expect the execution time of BUK to increase linearly with the problem size. As we see in Figure 8, the original version of BUK (without prefetching) suffers a large discontinuity in execution time once the problem no longer fits in memory (recall that our prototype has 64 MB of physical memory, roughly 48 MB of which is available to the application). In contrast, the prefetching version of the code suffers no such discontinuity—execution time continues to increase linearly. For this particular application, the prefetching version of the code outperforms the original code on all problem sizes, since even small problem sizes benefit from prefetching cold misses.[6] Hence this application exemplifies what we are attempting to accomplish with automatic I/O prefetching: programmers can write their code in a natural manner and still achieve good performance even on out-of-core problem sizes.

5 Related Work

Much related work has depended on the use of an explicit I/O interface by the programmer. On the OS side, this work includes the automatic detection of file access patterns in the file system [1, 10, 11, 12, 14, 15, 17], as well as the use of access patterns supplied directly by the application using an I/O type of interface [22, 26, 30, 4]. For compilers it involves analysis to move explicit I/O calls back and change them to asynchronous I/O calls instead [25]. While some of the OS policies developed may be useful in our environment, our goal is to avoid the use of explicit I/O entirely.

Of the file system prefetching techniques mentioned above, the work on Transparent Informed Prefetching (TIP) by Patterson et. al [22] is most relevant to our work in that hints provided by the application level are used by the operating system to optimize file prefetching and replacement. In fact, the cost model employed by TIP might be very useful for our memory manager. However, TIP targets applications which are written to use explicit I/O, and they depend on the programmer (rather than the compiler) instrumenting the code with hints. Also, their hints follow a very different model, where no concept of time is embedded in the hints, and hence their operating system support must be more complex than ours.

Compiling for out-of-core array codes tends to focus on two areas. The first is reordering computation to improve data reuse and reduce the total I/O required [3]. The second area is inserting explicit I/O calls into array codes [6, 13, 21, 29]. In general, the compilers are aided by extensions to the source code that indicate particular structures are out-of-core. In addition, some of the work specifically targets I/O performance for parallel applications [3], while we have achieved impressive speedups for even single-threaded applications.

[6] For BUK, it is more realistic to cold-start the application, since it must always read its input data set from disk.

We feel that compiler analysis that targets an I/O interface is limited by the alias analysis problem described earlier, and in general cannot be as aggressive as an algorithm that supports non-binding prefetching.

Other work has also been done in the area of prefetching for paged virtual memory systems. As for file systems, some of the work depends on the OS detecting patterns to initiate prefetching [8, 27]. These techniques suffer from the fact that some number of faults are required to establish patterns before prefetching can begin, and when the patterns change unnecessary prefetches will occur. Using application-specific knowledge to assist memory management policies was studied by Malkawi and Patel [18], however they only considered retaining needed pages in memory and did not consider prefetching.

The most relevant work to our study was conducted nearly twenty years ago by Trivedi [32], who looked at the use application access patterns extracted by a compiler to implement "prepaging". Although the interface to the OS is nearly identical, there are some significant differences. First, Trivedi's compiler analysis was restricted to programs in which blocking could be performed whereas previous studies on prefetching for caches have shown that many programs which can be prefetched cannot be blocked [20]. Thus, our approach is much more widely applicable. Second, improvements in compiler analysis enable us to be much more aggressive, allowing the prefetching of indirect references and other interesting structures. Third, we have found that the use of the run-time layer is essential to achieving good performance when the compiler must deal with symbolic loop bounds, whereas this component was missing from earlier work.

6 Future Work

We view this work as an encouraging first step, and we are currently extending our research in the following directions. We are implementing our requisite support within commercial operating systems so that future results can be collected on larger, more modern systems where I/O latency is expected to be even more of a problem. To address the challenges of multiprogrammed workloads—where multiple applications compete for shared resources—we are exploring new ways that the compiler and OS can cooperate so that applications can adapt their behavior to dynamically fluctuating resource availability, and we will make more extensive use of release operations to minimize memory consumption. Multiprocessors also provide interesting challenges, such as co-locating data (on disk) and computation within the same node to minimize network traffic. Page-based prefetching is applicable to domains other than disk I/O; for example, we are adapting our compiler technology to prefetch the page-sized chunks of data that are communicated between workstations in distributed shared memory (DSM) systems. Finally, we are investigating how to extend the scope of our work beyond array-based codes to also include pointer-based codes and other non-numeric applications.

7 Conclusions

This paper has demonstrated that with only minor modifications to current operating systems, we can enhance paged virtual memory to deliver high performance to out-of-core applications without placing any additional burden on the programmer. We have proposed and evaluated a fully-automatic scheme whereby the operating system and the compiler cooperate as follows: the compiler analyzes future access patterns to predict when page faults are likely to occur and when data is no longer needed, the operating system uses this information to manage I/O through non-binding *prefetch* and *release* hints, and a run-time layer interacts with the operating system to accelerate performance by adapting to dynamic behavior and minimizing prefetch overhead. We implemented our scheme in the context of a modern research compiler and operating system.

Our experimental results demonstrate that our scheme yields substantial performance improvements when we take unmodified, "in-core" versions of scientific applications and run them with out-of-core problem sizes. We successfully hid more than half of the I/O latency in all of the NAS Parallel benchmarks—in three cases, we eliminated over 98% of the latency. For five of the eight applications, this reduction in I/O stalls translates into speedups of roughly twofold, with two cases speeding up by threefold or more. Given these encouraging results, we advocate that commercial operating systems provide the modest support necessary for the *prefetch* and *release* operations.

8 Acknowledgments

We thank the entire Hurricane and Hector research teams for developing this experimental platform. In particular, we thank Ben Gamsa for answering countless questions about Hurricane's internal workings. We also thank Karen Reid, Eric Parsons, and Paul Lu for their help in collecting these results. This work is supported by grants from the Natural Sciences and Engineering Research Council of Canada. Todd C. Mowry is partially supported by a Faculty Development Award from IBM.

References

[1] M. Arunachalam, A. Choudhary, and B. Rullman. A prefetching prototype for the parallel file system on the Paragon. In *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 321–323, May 1995. Extended Abstract.

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, August 1991.

[3] R. Bordawekar, A. Choudhary, and J. Ramanujam. Automatic optimization of communication in out-of-core stencil codes. In *Proceedings of the 10th ACM International Conference on Supercomputing*, May 1996.

[4] Pei Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. Technical report, November 1995.

[5] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[6] T. H. Cormen and A. Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243, Dept. of Computer Science, Dartmouth College, November 1994.

[7] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, December 1995.

[8] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proceedings of the 1993 ACM-SIGMOD Conference on Management of Data*, pages 257–266, May 1993.

[9] J. M. del Rosario and A. Choudhary. High performance I/O for massively parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[10] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Conference Proceedings of the USENIX Summer 1994 Technical Conference*, pages 197–208, June 1994.

[11] A. S. Grimshaw and Edmond C. Loyot, Jr. ELFS: object-oriented extensible file systems. In *Proceedings of the First International Conference on Par-*

allel and Distributed Information Systems, page 177, December 1991.

[12] J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995.

[13] K. Kennedy, C. Koelbel, and M. Paleczny. Scalable I/O for out-of-core structures. Technical Report CRPC-TR93357-S, Center for Research on Parallel Computation, Rice University, November 1993. Updated August, 1994.

[14] D. Kotz and C. Schlatter Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January 1993.

[15] David Kotz and Carla Schlatter Ellis. Prefetching in file systems for MIMD multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):218–230, April 1990.

[16] O. Krieger and M. Stumm. HFS: A performance-oriented flexible file system based on building-block compositions. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 95–108, Philadelphia, May 1996.

[17] T. M. Kroeger and D. D. E. Long. Predicting file system actions from prior events. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 319–328, January 1996.

[18] M. Malkawi and J. Patel. Compiler directed management policy for numerical programs. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 97–106, December 1985.

[19] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994.

[20] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 27, pages 62–73, October 1992.

[21] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on data parallel machines. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 110–118, McLean, VA, February 1995.

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

[23] J. T. Poole. Preliminary survey of I/O intensive applications. Technical Report CCSF-38, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[24] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Department of Computer Science, Rice University, May 1989.

[25] A. L. Narasimha Reddy, P. Banerjee, and D. K. Chen. Compiler support for parallel I/O operations. In *ICPP91*, pages II:290–II:291, 1991.

[26] T.P. Singh and A. Choudhary. ADOPT: A dynamic scheme for optimal prefetching in parallel file systems. Technical report, NPAC, June 1994.

[27] I. Song and Y. Cho. Page prefetching based on fault history. In *USENIX Mach III symposium proceedings*, pages 203–213, April 1993.

[28] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, , and G. Peck. Scalability in the XFS file system. In *USENIX Technical Conference*, pages 1–14. Usenix, January 1996.

[29] R. Thakur, R. Bordawekar, and A. Choudhary. Compilation of out-of-core data parallel programs for distributed memory machines. In *IPPS '94 Workshop on Input/Output in Parallel Computer Systems*, pages 54–72. Syracuse University, April 1994.

[30] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. PASSION runtime library for parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.

[31] S. W. K. Tjiang and J. L. Hennessy. Sharlit: A tool for building optimizers. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[32] K.S. Trivedi. On the paging performance of array algorithms. *IEEE Transactions on Computers*, C-26(10):938–947, October 1977.

[33] R. C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9(1/2):105–134, 1995.

[34] Z. G. Vranesic, M. Stumm, R. White, and D. Lewis. The Hector Multiprocessor. *IEEE Computer*, 24(1), January 1991.

[35] D. Womble, D. Greenberg, R. Riesen, and S. Wheat. Out of core, out of mind: Practical parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–16, Mississippi State University, October 1993.