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

by

Angela K. Demke

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

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

Angela K. Demke

Master of Science

Graduate Department of Computer Science

University of Toronto

1997

In this thesis, we propose and evaluate a fully-automatic technique to manage the I/O requirements of out-of-core scientific applications. In our scheme, the compiler determines information on future access patterns and inserts *prefetch* and *release* operations to initiate I/O before the data is needed and to free pages that are no longer accessed. No modifications to the original application source code are needed to support our compiler analysis. The operating system provides support for the basic prefetch and release operations and interacts with a run-time layer to accelerate performance by minimizing prefetch overhead.

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 one application speeding up by over threefold.

Acknowledgements

The research presented in this thesis could not have been done alone, and I would like to express my sincere thanks to all who have contributed to its success.

First of all, I would like to thank my supervisor, Todd Mowry, for his guidance and support throughout the development of the research, and for the many helpful comments on the organization and presentation of this work. I would also like to thank my second reader, Michael Stumm, for further improving the caliber of this thesis. In addition, I wish to gratefully acknowledge the financial support provided by NSERC for this research.

I would also like to acknowledge the entire HECTOR and HURRICANE research teams for developing the research prototypes on which the ideas presented here were evaluated. (Special thanks go to HECTOR for staying alive and well over the last few years!) In particular, I would like to thank Orran Krieger for the guided tour of HURRICANE and the Hurricane File System, and for his many suggestions. In addition, I would like to thank Ben Gamsa for not only always knowing the answers to my questions about the HURRICANE kernel, but also for always being there to provide them. Finally, thanks go out to the many people who have reset HECTOR for me when I couldn't be there to do it for myself: Ben Gamsa, Karen Reid, Eric Parsons, Paul Lu, Steve Jantzi, and Todd Mowry.

I would also like to thank my fellow students at the University of Toronto for contributing to an enjoyable graduate experience. In particular, I would like to thank Ian Maione and Jin Lee for making my life a little more surreal.

Finally, I want to thank my partner, Joseph Brown, for putting up with my work habits and my entropic filing system, and for being wonderfully supportive throughout. I also want to thank my cocker spaniel, Smokey, for always making me smile.

Contents

Chapter 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, 26]. 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 [37].

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 [37]. 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.

The goal of our research is to preserve the abstraction of unlimited virtual memory for the programmer, while providing performance competitive with that which can be achieved by explicitly managing I/O. To motivate the approach we have taken, we first discuss the limitations of existing methods. Section 1.1 explains why existing virtual memory systems perform poorly for these types of applications. Section 1.2 explains why existing explicit I/O approaches are undesirable. Given the limitations of current methods, Section 1.3 gives a high-level description of our scheme and presents the key contributions of this work. The organization of the rest of this thesis is presented in Section 1.4.

1.1 The Problem with Paged Virtual Memory

The promise of virtual memory is to free programmers from concerns of the underlying system such as the size of memory and the available I/O devices. Sadly, existing systems have failed to deliver on this promise. 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 demonstrate later in Chapter 4. There are three primary reasons why virtual memory systems are unable to provide the best I/O performance: the timing of requests, the size of requests, and replacement decisions.

First, in a virtual memory system data is brought into memory in response to a *page fault*, which occurs when the process attempts to read a virtual memory location that is not in physical memory. Since the request is triggered when the data is needed, the process must stall for the full latency of a disk read. Typically, the operating system will give the CPU to another process when a page fault occurs because the latency is so large. If other runnable processes exist, this technique keeps the CPU busy and improves system throughput but it does not help the performance of the faulting process in any way.

Second, page faults typically result in only a single outstanding page-sized read request at a time for a given process. High bandwidth disk arrays cannot generally be exploited for such small requests (unless a single page is striped across multiple disks) since only a single disk is busy at any time. 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. 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 delay of a disk read.

Finally, when the amount of data accessed is greater than the size of physical memory, the operating system must decide what to evict to make room for new requests. Without knowledge of the application's future accesses the memory manager may re-allocate pages that are about to be used again, increasing the need to perform I/O and further degrading performance. Recently, Park, Scott and Sechrest have shown that customizing the replacement policy on a per-application basis can significantly improve the performance of out-of-core applications [24]. An additional concern from a resource management perspective is that the memory manager may allocate more memory to a process than it actually needs in order to delay making these replacement decisions. This may have a negative impact on competing processes.

Having shown why virtual memory delivers poor performance for out-of-core applications, we now discuss how explicitly managing I/O can overcome some of these difficulties and the new complications that explicit I/O introduces.

1.2 The Problem With Explicit I/O

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: (i) large requests make better use of available bandwidth; (ii) replacement decisions are made by the application; (iii) non-blocking I/O calls allow disk accesses to be overlapped with computation.

First, read and write requests can affect a large number of blocks in a single request. This is important in a system with high disk bandwidth (i.e., a large number of disks), since the parallelism in the underlying disk system can be exploited by large requests.

Second, with a read/write interface the application specifies the buffers that are the source or target of the file data. Hence, the application can ensure that data that will soon be accessed is buffered in memory. Also, the application can minimize the amount of memory used by ensuring that buffers that it will not access for a long time are freed, and if modified, written to disk.

Finally, asynchronous I/O interfaces are available that allow applications to issue I/O requests without blocking. By properly scheduling these requests, the application can overlap disk I/O with computation, in the best case allowing all the disk latency to be hidden.

While explicit I/O offers the potential for improved performance over paging, it unfortunately suffers from several disadvantages. The primary drawback 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

It seems clear that if a paged virtual memory system had full knowledge about an application's future access pattern, all the performance advantages of explicit I/O could be achieved without the disadvantages. In particular, the memory manager could make large requests to allow disk bandwidth to be efficiently exploited, manage the memory efficiently to minimize memory used and to avoid paging out data that will soon be accessed, and prefetch data that will soon be accessed to hide from applications the latency of I/O. Moreover, the memory manager could perform these optimizations while taking into account the total resources available, and while avoiding the system call and copying overhead of explicit I/O interfaces.

The application access patterns are best known at the application level, either as the result of programmer intervention or analysis performed by a compiler. Knowledge regarding resource usage is naturally available to the operating system. To unify the required information, there is thus a choice between communicating access patterns down to the operating system where resource usage is known or communicating resource usage up to the application level. Most (if not all) current research in improving I/O performance has focused on informing the operating system about application access patterns, casting the operating system in its traditional role as a mediator of resources between competing applications. The problem with this approach is that applications may have very complicated (but regular) access patterns which are difficult to convey to the operating system. Even if full disclosure is possible, the complexity required by the memory manager to make the best use of the given information may hurt performance for in-core applications that do not require such functionality. In contrast, the relevant information about available memory and disk usage can be transferred to the application level quite concisely, which is the approach we use.

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 uses information provided by the operating system to accelerate performance by adapting to dynamic behavior and minimizing prefetch overhead. In this thesis, 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. 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.

1.4 Organization of Thesis

This thesis is organized as follows. We begin in Chapter 2 by discussing how the compiler and the operating system can cooperate to automatically prefetch disk accesses for out-of-core applications. Chapter 3 provides a more detailed description of the analysis performed by the compiler and the changes from the original compiler algorithm for cache prefetching that were needed to support I/O prefetching. Next, in Chapter 4 we describe the framework used to evaluate our ideas and present our experimental results. Finally, in Chapter 5 we present related work, while Chapter 6 discusses our conclusions and directions for future investigation.

Chapter 2

Automatically Tolerating I/O Latency

This chapter 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, 18, 29]. Roughly speaking, one can always increase the I/O bandwidth by purchasing additional disks.

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 a new page that is 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 operating system 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 operating system 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 references that commonly occur in sparse-matrix applications would be extremely difficult for the operating system to predict), and would require adding additional complexity to the operating system, which is something we wish to avoid.[1] 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 / Operating System Interface

Given that the compiler will be extracting and passing access pattern information to the operating system, 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 operating system (so that it can be readily incorporated into an existing commercial operating system), 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 operating system 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

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

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 operating system, 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 operating system. For example, if the compiler indicates that the program will be streaming through a large array, it is not helpful if the operating system 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 operating system 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* [21]. 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,

```
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 2.1: Example illustrating the importance of non-binding prefetches .

it is a binding prefetch. To illustrate this problem, consider the code in Figure 2.1(a). If we use the read/write interface, we might generate code similar to Figure 2.1(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 2.1(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 operating system to perform an I/O access. Instead, we would prefer to give the operating system the flexibility to drop requests if doing so might achieve better performance, given 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 operating system 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 [21, 22]. 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 Chapter 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[2], 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 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 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 operating system, 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 [21, 22]. Conceptually, prefetching from disks into main memory entails moving down one level in the memory hierarchy. To accommodate this transition, 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*. More details on each of these steps and the changes needed to properly support I/O prefetching are described in Chapter 3.

[2] 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.

2.4 Operating System / Run-Time Layer Interaction

The next two subsections describe the run-time layer that we introduce to reduce the overhead of unnecessary prefetches, and the support provided by the operating system.

2.4.1 The Run-Time Layer

The run-time layer tracks pages that are expected to be in memory by using a bit vector to construct a map of the application's virtual memory space. Each bit in the vector represents one or more contiguous virtual memory pages, with a set bit indicating that the corresponding page is in physical memory. 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 operating system, thus substantially reducing overhead. If a block of contiguous pages are specified in the prefetch request, we check each page until one is found that is not in memory, or the end of the block is reached. When a page is found that needs to be prefetched, the run-time layer issues a request to the operating system for the missing page and all subsequent pages in the block. In this way, at most one system call is required for a block prefetch.

Although it would be possible for the run-time layer to approximate the state of main memory entirely at the user level by setting bits for prefetched pages and clearing bits for released pages, the run-time layer would be unaware of pages brought into memory by page faults, or reclaimed by the operating system. To give the run-time layer a more accurate view, we instead make the bit vector shared with the operating system, allowing bits to be set or cleared by either level as pages are transferred in or out of memory.

2.4.2 Operating System Support

To support compiler directed prefetching, the operating system needs to be able to respond to the prefetch and release requests issued by the application. For a prefetch request, the operating system allocates a memory page from the list of free pages to hold the requested data and issues an asynchronous read request to the file system before returning control to the application. In the event that there is no free memory available the operating system is permitted to drop the prefetch request. It is reasonable to drop the request because it is expected that prefetching applications are attempting to limit their memory consumption by releasing pages that are not currently needed. If all pages are in use at some time, choosing to evict one to satisfy a prefetch request could hurt performance. For a release request the operating system unmaps the specified page and places it on the free list, scheduling a write if necessary. Functionality for these operations is easy to add since most operating systems already support asynchronous requests to the file system and need to be able to unmap pages of memory for existing memory management activities.

A more interesting issue is having the operating system actively share a data structure (i.e. the bit vector) with the user level. The operating system agrees to provide a user-level process with a single page of physical memory that can be used as a map of the process's virtual memory space. Since only a single page will be provided, the *granularity* of the bit vector must be established. For instance if the page size is 4 KB and each bit represents a single page, then the bit vector is capable of tracking 2^{15} pages of virtual memory or 134 MB of data. If the application accesses more data than can be represented with a single bit per page, then each bit must represent multiple pages and both the operating system and the run-time layer must agree on the granularity to use. In general, either the operating system or the run-time layer could decide on the appropriate granularity and inform the other layer of the choice. In our system, the run-time layer uses information about the size of mapped files to set the granularity of the bit vector. When a prefetching application is executed, the run-time layer binds a page-sized region of memory to a fixed virtual address to use as the bit vector. Next, a system call passes the starting virtual address and the granularity of the bit vector to the operating system, which records the physical page corresponding to the specified virtual address and the granularity in the process's address space descriptor. At this point, the operating system and the run-time layer have agreed on which page will be shared and how it will be used to map the process's virtual address space. The operating system can now set bits in the vector whenever the process page faults, and clear them whenever pages belonging to the process are unmapped.

The granularity of the bit vector needs to be considered by both the run-time layer and the operating system, not only to ensure that the right bits are set, but also to ensure that the the bit vector remains useful for the purposes of the run-time layer. For example, if the compiler schedules a prefetch request for a single page and the run-time layer sets the corresponding bit before fetching the page, then it will appear as if all pages in the same *group* (i.e. all pages represented by the same bit) are already in memory. The result is that a prefetch request for another page in that group will be filtered out by the run-time layer. The same problem can occur when the operating system sets bits for pages brought into memory through page faults. In our implementation we handle granularities greater than a single page as follows. The run-time layer asks the operating system to prefetch *all* the pages in a group whenever a prefetch is needed for *any* page in that group, thus preserving the notion that the pages are in memory if the corresponding bit is set. In turn, when a single page is brought into memory due to a page fault the operating system does not set the bit, allowing the run-time layer to still issue prefetch requests for the other pages in the group.

Chapter 3

The Compiler Algorithm

In this chapter we provide a more detailed description of our compiler algorithm for generating prefetch and release requests. The bulk of this algorithm is a straightforward extension of one that was developed earlier for prefetching cache-to-memory misses in dense-matrix and sparse-matrix codes [21, 22]. Essentially, we simply move down one level in the memory hierarchy to prefetch data from disks into main memory. Thus, the basic input parameters that describe the cache size, line size, and miss latency in the original algorithm are changed to reflect memory capacity, page size, and page fault latency, respectively.

Additional modifications to the original algorithm are needed for the following reasons:

- We need to generate *release* requests to identify pages that are no longer needed by the application.
- The cost of issuing a prefetch request is much greater for I/O prefetching since operating system interaction is required. We seek to amortize the system call overhead by requesting multiple pages with a single *block prefetch* request whenever possible.
- A memory page contains many more data items than a cache line. While this may seem obvious, the implications for how the compiler schedules prefetches need to be carefully considered. In particular, different techniques for splitting loops and pipelining prefetches must be applied.

Most of the required changes relate to how prefetches are scheduled, however it is difficult to understand why certain references need to be handled in a particular manner without first seeing how the compiler determines what needs to be prefetched. Hence, we begin in Section 3.1 with a description of how the compiler uses *locality analysis* to predict when misses (i.e. page faults) are likely to occur. Most of the work in this section has been presented previously in Mowry's thesis on cache prefetching [21] and is reproduced here for completeness and ease of reference. Next, Section 3.2 discusses how the compiler first isolates these faulting instances through *loop splitting* techniques, and then schedules prefetches early enough using *software pipelining*.

3.1 Locality Analysis

The goal of the *locality analysis* step of the prefetching algorithm is to identify precisely which references are likely to incur page faults. To accomplish this, it is necessary to determine both when data is reused, and whether that data is expected to remain in memory between uses. The locality analysis step is fundamentally unchanged from that developed for cache prefetching, however it is important to understand how the compiler decides what to prefetch to understand why some references are "missed".

It should be noted that locality analysis is only applied to "direct" array references (i.e. `A[i]`) and not to "indirect" references (i.e. `A[B[i]]`), because it is impossible to determine the index values, `B[i]`, at compile-time. In the best case, all of the values may be the same, and the reference would only need to be prefetched on the first access. At the other end of the spectrum, each index may point to a different page and the reference would need to be prefetched all the time. In general, we choose to always prefetch indirect references (since the benefit of potentially eliminating I/O stalls is so great) and rely on run-time techniques to reduce the overhead when they are unnecessary.

It is also important to note that locality analysis is applied to each set of nested loops independently. Thus, the type of reuse shown in Figure 3.1 cannot be discovered by the current compiler algorithm. The key ideas needed for locality analysis are introduced in Section 3.1.1, and Section 3.1.2 uses an illustrative example to show how these ideas can be expressed in terms of loop iterations. With this framework, the remaining three subsections describe each step of the locality analysis algorithm in detail.

Example Code

```
for (i = 0; i < 100; i++)
  A[i] = i;

for (i = 0; i < 100; i++)
  A[i] = A[i] * i;
```

Figure 3.1: Example of reuse not identified by compiler.

3.1.1 Fundamental Concepts

A data item has *reuse* if the same data item is referenced multiple times. Reuse is thus an intrinsic property of a given data access pattern. In contrast, *locality* only results when subsequent references find the data item still in memory. Hence, it is a function of the size of memory, the volume of data accessed between reuses, and the page replacement policy used by the operating system. Since the operating system's replacement policy is beyond the compiler's ability to analyze, we simply assume that a page is likely to be replaced if the amount of data accessed between reuses is greater than the size of physical memory. This assumption would be true for a strict LRU replacement policy. If memory were unlimited, then reuse and locality would be identical; in reality, the references with data locality are a subset of those with data reuse.

To properly identify what needs to be prefetched, we must distinguish three types of data reuse (and three corresponding types of locality), each of which needs to be handled in a different manner. *Temporal reuse* occurs when a particular reference accesses exactly the same data location in different iterations. *Spatial reuse* occurs when a particular reference accesses different data locations found on the same page. *Group reuse* occurs when different references access data locations found on the same page.

Given the relationship between reuse and locality, the locality analysis algorithm is comprised of three steps:

1. Discover the intrinsic data reuses within a loop nest through *reuse analysis*. This would be equivalent to solving the locality analysis problem if memory were unlimited.

2. Given that we have a *finite* memory, determine the set of reuses that actually result in locality. This is accomplished by computing the *localized iteration space*, which is the set of nested loops that access less data than the specified memory capacity. Data locality is then computed by intersecting the intrinsic data reuses with the localized iteration space. i.e.

$$\text{Data Reuse} \cap \text{Localized Iteration Space} \Rightarrow \text{Data Locality}$$

3. Express the data locality for each reference in terms of a *prefetch predicate*, which is a logical predicate that is true during each dynamic iteration when the reference is expected to incur a page fault. These predicates are used during scheduling to split loops, statically isolating the faulting references.

The first two steps produce a mathematical description of locality in a *vector space* representation. The third step translates this description into a representation which is more directly applicable to the problem of scheduling the required prefetches. To gain an intuition for the concepts captured by the vector space notation, we present an example in Section 3.1.2, before discussing the details of each step in Sections 3.1.3, 3.1.4, and 3.1.5, respectively.

3.1.2 An Example

The types of reuse that can be identified using *reuse analysis* are shown in Figure 3.2(a). For this example, assume the data are stored in row-major order and that each memory page holds two array elements. These parameters are chosen for illustrative purposes only, and are unrealistically small. The iterations that first touch a new page, and those that reuse the same page are shown graphically in Figure 3.2(b) using *iteration space* plots. In these plots, the horizontal axis represents the `j` loop, while the vertical axis represents the `i` loop. In other words, each row of the plots corresponds to a single iteration of the `i` loop, while each node within a row corresponds to a single iteration of the `j` loop. Hence, each node represents the result of the data access in a particular iteration. During the execution of the corresponding code, the nodes within a row would be visited from left to right, and the rows would be visited from bottom to top.

(a) Example Code

```
for (i = 0; i < 3; i++)
  for (j = 0; j < 8; j++)
    A[i][j] = B[j][0] + B[j+1][0];
```

(b) Data Reuse

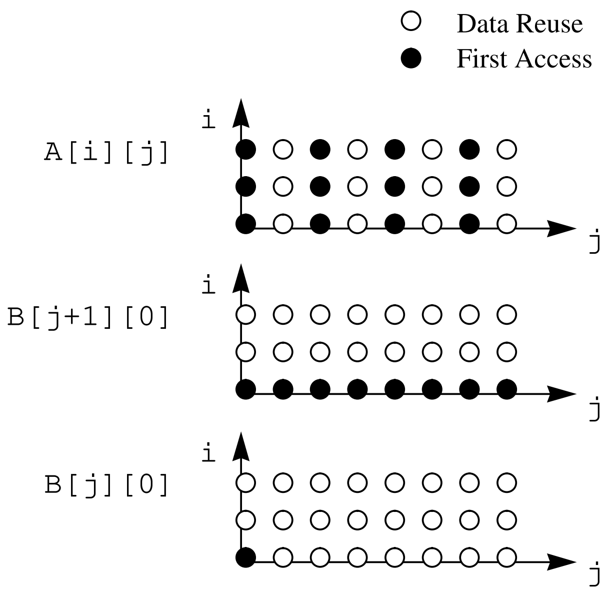


Figure 3.2: Data reuse example.

The A[i][j] reference in Figure 3.2 accesses each data element exactly once, traversing the rows of the matrix along the inner loop. Since each page contains two array elements, a new page is touched on every other iteration of the inner loop as page boundaries are crossed. The iterations that first touch new pages are illustrated in the first plot of Figure 3.2(b). This reference exhibits *spatial reuse.*

The B[j+1][0] reference traverses the columns of the matrix along the inner loop, causing each reference to touch a different page during the first iteration of the j loop. However, exactly the same locations are used again on subsequent iterations of the i loop, resulting in *temporal reuse* for this reference. This effect is illustrated in the second plot of Figure 3.2(b).

The B[j][0] and B[j+1][0] references provide an example of *group reuse.* In this case, the B[j][0] reference uses the same data locations first accessed by the B[j+1][0] reference during the previous iteration of the `j` loop. Thus, the `B[j][0]` reference only touches a new page during the first `j` loop iteration. The third plot of Figure 3.2 shows this effect. For references with group locality, we need to identify the *leading reference*, which is the reference that accesses new data first and thus will incur most of the page faults. We also need to identify the *trailing reference*, which is the last reference to touch the data. Only the leading reference is considered when scheduling prefetches, while the trailing reference is used for releases. In this example, `B[j+1][0]` is the leading reference and `B[j][0]` is the trailing reference.

For loops as small as the ones shown in this example, it is probable that the reuse would also result in locality. Section 3.1.3 describes how these types of reuse are identified, while Section 3.1.4 shows how the compiler determines when reuse leads to locality.

3.1.3 Reuse Analysis

Since locality can only occur if there is reuse, the first step in locality analysis is determining the intrinsic data reuse through *reuse analysis*. Reuse analysis attempts to discover those instances of array accesses that refer to the same page of memory. The difference is that while reuse is an inherent property of code, locality also depends on the ability of memory to retain data. Therefore if we had an infinitely large memory, which would retain data perfectly, reuse would be equivalent to locality.

A key simplification of reuse analysis is that rather than trying to precisely compute the sets of iterations (i.e. actual loop index values) that use the same data, which is prohibitively expensive, we instead express the intuitive notion that reuse is carried by a specific loop with the following mathematical formulation. We represent an n-dimensional loop nest as a polytope in an n-dimensional iteration space (i.e. a finite convex polyhedron bounded by the loop bounds), with the outermost loop represented by the first dimension in the space. We represent the shape of the set of iterations that use the same data by a *reuse vector space*[36]. The remainder of this subsection describes how this mathematical representation is used to compute temporal, spatial, and group reuse.

Temporal Reuse

Since temporal reuse occurs whenever a given reference accesses the same location in multiple iterations, we can isolate these cases by solving for whenever the array indexing functions yield identical results given different loop indices. To facilitate this task, we represent an array indexing function $\vec{f}(\vec{\imath})$ which maps n loop indices into d array indices, where n is the depth of the loop nest and d is the dimensionality of the array, as

$$\vec{f}(\vec{\imath}) = H\vec{\imath} + \vec{c}$$

where H is a d x n linear transformation matrix, $\vec{\imath}$ is an n-element iteration space vector, and $\vec{c}$ is a d-element constant vector. For example, the three array references in Figure 3.2 would be written as

$$\begin{aligned}
\texttt{A[i][j]} &= \texttt{A}\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right), \\
\texttt{B[j][0]} &= \texttt{B}\left(\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right), \text{ and} \\
\texttt{B[j+1][0]} &= \texttt{B}\left(\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right).
\end{aligned}$$

Given this representation, temporal reuse occurs between iterations $\vec{\imath}_1$ and $\vec{\imath}_2$ whenever $H\vec{\imath}_1 + \vec{c} = H\vec{\imath}_2 + \vec{c}$, i.e. when $H(\vec{\imath}_1 - \vec{\imath}_2) = \vec{0}$. Rather than worrying about individual values of $\vec{\imath}_1$ and $\vec{\imath}_2$, we say that reuse occurs along the direction vector $\vec{r}$ when $H(\vec{r}) = \vec{0}$. The solution to this equation is the *nullspace* of H, which is a vector space in $\mathcal{R}^n$ (i.e. each vector has n components).

To make this analysis more concrete, consider the `B[j+1][0]` reference from our example in Figure 3.2. This reference accesses the same location in iterations (i_1, j_1) and (i_2, j_2) whenever

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \text{ or}$$

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

This equation is true whenever $j_1 = j_2$, and regardless of the difference between i_1 and i_2. In our vector space representation, we would say that temporal reuse occurs whenever the difference between the two iterations lies in the nullspace of $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$, that is, span $\{(1,0)\}$. We refer to this vector space as the temporal reuse vector space. This mathematical approach succinctly captures the intuitive concept that the direction of reuse of `B[j][0]` lies along the outer loop. More complicated access patterns can also be expressed using the same analysis.

Spatial Reuse

Computing spatial reuse requires a slight variation on how we compute temporal reuse. Assuming the data are stored in row major order (an assumption we make without loss of generality), accesses to the same page will only occur when the same row is accessed.[1] In addition, the row index expressions must be different, but still fall within the size of a cache line. We can test for all of these conditions as follows.

Two different iterations access the same row whenever all but the row index are equivalent. This is in contrast with temporal reuse, where *all* indices, including the row, must be equivalent. To extract this *spatial reuse vector space*, we simply replace the last row in H with zeros to create H_S, and solve for the nullspace of H_S. For example, consider the `A[i][j]` reference in Figure 3.2, where $H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, and therefore $H_S = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$. The resulting nullspace of $\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ is span $\{(0,1)\}$, which indicates that the same row of `A[i][j]` is accessed along the inner loop.

To check whether *different* elements are being accessed within the same row, we compare whether the temporal and spatial reuse vector spaces are identical. This can occur since reusing the same data item is a degenerate case of reusing the same page (i.e. the nullspace of H is a subset of the nullspace of H_S). If the temporal and spatial reuse vector spaces *are* identical, then there is strictly temporal reuse—if they differ, then there is spatial reuse along the vectors that are unique to the spatial reuse vector space.

[1] If the data are laid out in *column* major order, then accesses to the same page can only occur when the same *column* is accessed.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    foo(C[i],C[j]);
    bar(D[2i][j]), D[2i+1][j]);
```

Figure 3.3: Example of references without exploitable group reuse.

For example, the `A[i][j]` reference in Figure 3.2 has a temporal reuse vector space of span $\{\}$ (i.e. there is no temporal reuse), and a spatial reuse vector space of span $\{(0,1)\}$; therefore unique elements within the row are accessed along span $\{(0,1)\}$ (i.e. the inner loop). For the `B[j][0]` reference, however, the temporal and spatial reuse vector spaces are both span $\{(1,0)\}$, and therefore there is only temporal reuse.

Once we identify accesses to different elements within the same row, the final step is to check whether the stride is less than the page size. If so, then the reference has spatial reuse.

Group Reuse

For reuse among different array references, Gannon *et al.* observe that data reuse is exploitable only if the references are *uniformly generated*; that is, references whose array index expressions differ in at most the constant term[10]. For example, references `B[j][0]` and `B[j+1][0]` in Figure 3.2 are uniformly generated, while references `C[i]` and `C[j]` in Figure 3.3 are not. In the former case, `B[j][0]` is accessing data brought into the cache by `B[j+1][0]` during the previous `j` iteration, making it very likely that this reuse will result in locality. In the latter case, only iterations near the diagonal (i.e. when `i` $=$ `j`) are likely to have exploitable reuse. Thus we will only consider group reuse among sets of uniformly generated references.

Although uniformly generated references are the likely candidates for group reuse, it is still possible that they never access the same data. For example, the `D[2i][j]` and `D[2i+1][j]` references in Figure 3.3 are uniformly generated, but never touch the same data since `D[2i][j]` only accesses even rows while `D[2i+1][j]` only accesses odd rows of matrix `D`. To exclude such cases, we check whether a *particular solution* to the common transformation matrix H exists that yields the constant difference between the two array index functions. We express this mathematically by saying that two distinct references `A[`$H\vec{\imath}+\vec{c}_1$`]` and `A[`$H\vec{\imath}+\vec{c}_2$`]` access the same data if and only if

$$\exists \vec{r} : H\vec{r} = \vec{c}_1 - \vec{c}_2. \tag{3.1}$$

For example, the two references to the D matrix in Figure 3.3 would be represented as

$$\begin{aligned} \texttt{D[2i][j]} &= \texttt{D}\left(\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right), \text{ and} \\ \texttt{D[2i+1][j]} &= \texttt{D}\left(\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right). \end{aligned}$$

These two references access the same data if an integer solution (i_r, j_r) exists to

$$\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i_r \\ j_r \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

However, there is no integer solution in this case, since there is no integer i_r such that $2i_r = 1$. In contrast, the `B[j][0]` and `B[j+1][0]` references in Figure 3.2 access the same data since $(i_r, j_r) = (0, 1)$ is one particular solution to

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} i_r \\ j_r \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

While equation (3.1) specifies cases where distinct references access the same *data item*, we are also interested in cases where distinct references access the same *page*.

To detect all of these group reuse cases, we modify equation (3.1) slightly by replacing the last rows in H, $\vec{c}_1$, and $\vec{c}_2$ with zeros to form H_S, $\vec{c}_{S,1}$, and $\vec{c}_{S,2}$, respectively. We therefore say that two distinct references `A[`$H\vec{\imath}+\vec{c}_1$`]` and `A[`$H\vec{\imath}+\vec{c}_2$`]` have group reuse if and only if

$$\exists \vec{r} : H_S\vec{r} = \vec{c}_{S,1} - \vec{c}_{S,2}, \tag{3.2}$$

and also provided that the constant difference between the last rows of $\vec{c}_1$ and $\vec{c}_2$ is less than the cache line size divided by the element size. In reality, the alignment of the two references with respect to the page boundaries may also be a concern. In the worst case, adjacent references may always straddle page boundaries, potentially never reusing the same cache lines. In our implementation, we account for this probabilistically by assuming that two references within half a page of each other probably fall within the same page.

Having demonstrated how the various types of reuse can be computed, the next step in the algorithm is determining when reuse actually results in a locality benefit, given that memory is *finite* rather than *infinite*. To make this distinction, the following subsection introduces the concept of the *localized iteration space*.

3.1.4 Localized Iteration Space

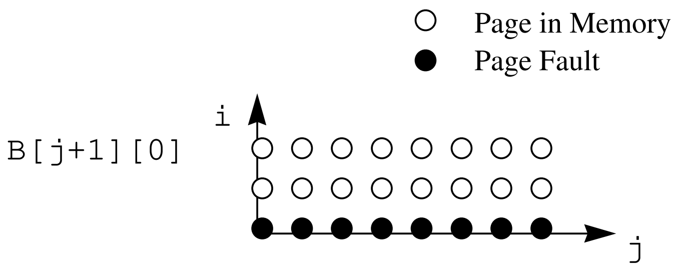
The *localized iteration space* is defined as the set of innermost loops that access less data in a single iteration than the available memory capacity. This definition implies that the localized iteration space includes only those loops for which reuse can result in locality, since if the volume of data accessed in a single iteration is greater than the size of memory, then reuse that occurs in subsequent iterations is unlikely to find the data in memory. The total aggregate data traffic across a particular loop nest is computed by taking the amount of data accessed by each reference and multiplying it by the number of times that reference is seen (i.e. the number of loop iterations), subject to the type of reuse that the reference may have.

The localized iteration space is represented as a vector space so that it can be directly compared with the vector space representation of data reuse, facilitating the computation of data locality.

To illustrate these concepts, we consider the example shown in Figure 3.4. For this example, we will assume that we have 500 memory pages, that each page contains two array elements, and that the data are laid out in row-major order. The code for Figure 3.4(a) is identical to that in Figure 3.2, however in Figure 3.4 the number of `j` loop iterations has been increased from eight to 10,000. Iteration space plots are shown only for the `B[j+1][0]` reference, which has temporal reuse along the outer loop. The localized iteration space for this example is computed by comparing the number of pages accessed by all references against the number of pages in memory.

(a) Few Iterations in Inner Loop

```
for (i = 0; i < 3; i++)
  for (j = 0; j < 8; j++)
    A[i][j] = B[j][0] + B[j+1][0];
```



(b) Many Iterations in Inner Loop

```
for (i = 0; i < 3; i++)
  for (j = 0; j < 10000; j++)
    A[i][j] = B[j][0] + B[j+1][0];
```

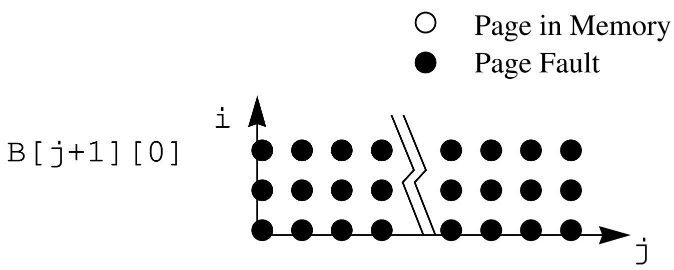


Figure 3.4: Example of how loop iteration counts affect locality.

Consider first the `A[i][j]` reference, which has spatial reuse along the inner loop (since each page contains two array elements) and no reuse along the outer loop (since different columns are being accessed). A single iteration of the `j` loop will bring one page into memory. To find the number of pages accessed in a single iteration of the outer loop, we simply multiply the number of pages accessed in an inner loop iteration and divide by the spatial reuse factor. Thus, for a single iteration of the outer loop in Figure 3.4(a) the `A[i][j]` reference brings $\frac{8*1}{2} = 4$ pages into memory, whereas in

Figure 3.4(b) $\frac{10000*1}{2} = 5000$ pages are brought into memory. To find the number pages accessed in the entire loop nest by this reference, we simply multiply the pages accessed in a single iteration by the number of iterations (since `A[i][j]` has no reuse along the outer loop).

For the `B[j][0]` reference, which has group reuse with the `B[j+1][0]` reference, only the first iteration of the inner loop (when `j = 0`) causes a new page to be brought into memory. Hence, for both examples in Figure 3.4, this reference contributes a single page to the total data traffic across the entire loop nest. In practice, because this reference has group reuse its contribution to the total data traffic is expected to be small and the reference is ignored.

Finally, the `B[j+1][0]` reference, which has temporal reuse along the outer loop, touches a distinct page during each iteration of the inner loop with no savings due to spatial reuse. The number of pages accessed in a single iteration of the outer loop is thus the same as the number of times the inner loop is executed (i.e. 8 for the code in Figure 3.4(a) and 10,000 for the code in Figure 3.4(b)). Since this reference has temporal reuse along the outer loop, the total number of pages accessed in the entire loop nest does not increase any further.

By summing each reference's contribution to the total data traffic, we can now determine whether each loop lies within the localized iteration space or not. For the code in Figure 3.4(a), a single iteration of the `j` loop brings at most three pages into memory (one for the `A[i][j]` reference, one for the `B[j][0]` reference, and one for the `B[j+1][0]` reference). Since this is much less than our memory capacity of 500 pages, we can conclude that the `j` loop is within the localized iteration space. Similarly, a single iteration of the `i` loop brings only 13 pages into memory (four for `A[i][j]`, eight for `B[j+1][0]` and one for `B[j][0]`). For this example, both loops are within the localized iteration space, which we would represent using vector space notation as span $\{(1,0),(0,1)\}$. In contrast, for the code in Figure 3.4(b), a single iteration of the inner loop brings again brings only three pages into memory, while a single iteration of the outer loop brings 15,001 pages into memory (5,000 for `A[i][j]`, 10,000 for `B[j+1][0]` and one for `B[j][0]`). In this case, only the inner loop is within the localized iteration space, which we represent as span $\{(0, 1)\}$.

Once the localized iteration space has been computed and expressed using vector space notation, computing locality is simply a matter of intersecting the reuse vector space with the localized iteration space, i.e.

$$\text{Reuse Vector Space} \cap \text{Localized Iteration Space} \Rightarrow \text{Locality Vector Space.}$$

Both spatial and temporal locality can be computed in this manner, however group locality is treated in a slightly different fashion. As discussed in section 3.1.3, group reuse exists only if there is a *particular solution* to the difference between the array index equations of two references. To determine whether group locality exists, we check to see if the particular solution is within the localized iteration space. i.e.

$$\vec{r}_p \in \text{Localized Iteration Space} \Rightarrow \text{Group Locality.}$$

A set of references with group locality are all placed in the same *equivalence class*, indicating that they can all be treated as a single reference. References with group reuse but not group locality are placed in different equivalence classes. The particular solution is also used to identify the *leading reference* in an equivalence class, which is the reference that first touches new data (and will thus incur the bulk of the page faults). Similarly, the *trailing reference*, which is the last reference to touch the data, is identified to generate release requests.

In practice, a number of considerations complicate the computation of data locality. First, symbolic loop bounds make it difficult to determine the exact amount of data accessed in a loop nest. Aggressive constant propagation can resolve some of these cases, but in other instances the compiler must simply assume unknown loop bounds to be either small (always localized) or large (never localized). Second, the actual amount of memory available at run-time may be quite different from that specified at compile-time, due to the resource demands of other applications. This problem can be handled by under-estimating the amount of available memory at compile-time and relying on the run-time layer to reduce the overhead of unnecessary prefetches generated by this approach.

We turn now to the final step in the locality analysis algorithm - converting the vector space representations of data locality into prefetch predicates that can be used for scheduling the prefetches.

3.1.5 The Prefetch Predicate

The purpose of constructing prefetch predicates is to associate a logical predicate which evaluates to "True" whenever a reference is expected to incur a page fault. These predicates can then be used to split loops such that iterations where the predicate has the same value are grouped together, and faulting iterations are isolated. Different predicates are constructed corresponding to each type of locality that a reference may have. For instance, if a reference has no locality, it is expected to page fault on every iteration and the associated prefetch predicate is simply "True". A reference with temporal locality with respect to a particular loop will only page fault during the first iteration of that loop, thus the associated predicate is "`loop_index` = 0". A reference that has spatial locality with respect to a given loop will only page fault when page boundaries are crossed. If the number of data elements in a page is l, this will occur whenever "`loop_index` mod l = 0", which is the prefetch predicate for spatial locality. Finally, references with group locality that are not the leading reference are rarely expected to incur page faults, thus the prefetch predicate is set to "False" indicating that these references should never be prefetched.

To calculate the overall prefetch predicate for a reference within a multi-level loop, we first construct the predicate with respect to each surrounding loop and then take the conjunction of all these predicates. Figure 3.5(b) shows an example of how prefetch predicates are constructed for the given code. For instance, the `A[i][j]` reference has spatial locality with respect to the `j` loop, and each page contains two array elements, yielding a predicate of "`j` mod 2 = 0". This reference has no locality with respect to the `i` loop, giving a predicate of "True". Taking the conjunction of these two predicates, the overall prefetch predicate for `A[i][j]` is simply "`j` mod 2 = 0". The other two references are handled in a similar fashion and the results are shown in Figure 3.5.

The reasoning applied to the locality vector space representation to construct prefetch

(a)

```
for (i = 0; i < 3; i++)
  for (j = 0; j < 100; j++)
    A[i][j] = B[j][0] + B[j+1][0];
```

(b)

Reference	Locality	Prefetch Predicate
`A[i][j]`	$\begin{bmatrix} \mathtt{i} \\ \mathtt{j} \end{bmatrix} = \begin{bmatrix} \text{none} \\ \text{spatial} \end{bmatrix}$	$(\mathtt{j} \bmod 2) = 0$
`B[j+1][0]`	$\begin{bmatrix} \mathtt{i} \\ \mathtt{j} \end{bmatrix} = \begin{bmatrix} \text{temporal} \\ \text{none} \end{bmatrix}$	$\mathtt{i} = 0$
`B[j][0]`	$\begin{pmatrix} \text{Group with} \\ \mathtt{B[j+1][0]} \end{pmatrix}$	*False*

Figure 3.5: Example of how prefetch predicates are constructed.

predicates can also be applied to determine when data is no longer needed and can thus be released. In general, the instances when we want to release pages are closely related to when we need to prefetch them. For example, a reference with no locality needs to be prefetched *before* every use, and can be released *after* each use (since it will not be used again). A reference with temporal locality needs to be prefetched before the *first* loop iteration, and can only be released after the *last* iteration. For spatial locality, we prefetch before the first reference to a new page, and release after the last reference to that page. For references with group locality, we need to prefetch ahead of the leading reference and release after the trailing reference.

After constructing the prefetch predicates, the compiler needs to schedule prefetches for the references that are expected to incur page faults (i.e. those for which the predicate evaluates to true), and releases for data that is no longer being used. The next section describes how loop splitting techniques are applied to transform loops and isolate faulting references, and how software pipelining is used to schedule prefetches the right amount of time before the expected reference.

3.2 Scheduling Prefetches

Most of the changes made to the original compiler algorithm for prefetching [21] are related to how loops are split and how prefetch and release operations are scheduled. The objectives of the scheduling phase are twofold: first, we want to issue prefetches early enough that the requested data are found in memory when they are needed; second, we want to minimize the overhead associated with issuing prefetch and release requests. To minimize overhead, we apply two techniques. First, we isolate faulting references by splitting loops to avoid introducing conditional statements in inner loops. This technique was developed for cache prefetching and remains an important optimization. Second, we attempt to minimize the number of times the application/system boundary is crossed, since the cost of making a system call contributes greatly to the software overhead of issuing prefetches.

In the following subsections, we discuss how prefetch and release operations are scheduled, and emphasize the changes required for I/O prefetching.

3.2.1 Loop Splitting Techniques

The purpose of loop splitting is to statically isolate all iterations in which the prefetch predicate for a particular reference have the same value. Once this has been done, we either always prefetch at a given static point in the code, or never prefetch Hence, there is no need to evaluate the prefetch predicate to determine when to prefetch. Just as a different predicate was constructed for each type of locality, a different splitting technique is applied for each type of predicate.

First, if the prefetch predicate is either "True" or "False", no transformation is required since the loop is already in a form where we either always prefetch or never prefetch.

For temporal locality, the predicate is "`loop_index` $= 0$"; we need to prefetch during the first iteration and release during the last iteration. In this case, we would isolate the instances where we need to prefetch and the instances where we need to release by *peeling* the first and last iterations respectively. An example of the effect of this transformation

(a) Code Before Peeling

```
for (i = 0; i < n; i++) {
  if (i == 0)
    f(i);
  g(i);
  if (i == n-1)
    h(i);
}
```

(b) Code After Peeling

```
f(0);
g(0);
for (i = 1; i < n-1; i++) {
  g(i);
}
g(n-1);
h(n-1);
```

Figure 3.6: Example of peeling the first and last iterations of a loop

on a generic loop is shown in Figure 3.6.

For spatial locality, the predicate is "`loop_index` mod $l = 0$"; we need to prefetch and release once every l iterations. With cache prefetching, the preferred technique was to replicate the body of the loop l times, using *loop unrolling*. However, for I/O prefetching l may be large (typically several hundred elements), and the preferred technique is to *strip-mine* the loop by a factor of l, since replicating the loop body several hundred times is unfeasible. Examples of both unrolling and strip-mining are shown in Figures 3.7 and 3.8, respectively. There is one additional complication that arises when prefetching references with spatial locality from disk into memory. Since l is typically quite large, it is possible that a given loop nest (and in some cases multiple loop nests) access less than a full page of data. In contrast, with cache prefetching it was extremely unlikely that a loop would access less data than a cache line. In these cases, the loop iteration count is less than the strip-mining factor, l, thus making strip mining unfeasible. We also need to be more careful about scheduling for these types of references, as will be discussed in more detail in the following section.

These loop splitting techniques can be applied recursively to nested loops to handle multiple prefetch predicates. One possible area of concern is the amount of code expan-

(a) Code Before Unrolling

```
for (i = 0; i < n; i++) {
  if ((i mod 4) == 0)
    f(i);
  g(i);
  if ((i mod 4) == 0)
    h(i);
}
```

(b) Code After Unrolling

```
for (i = 0; i < n; i+=4) {
  f(i);
  g(i);
  h(i);
  g(i+1);
  g(i+2);
  g(i+3);
}
```

Figure 3.7: Example of unrolling a loop by a factor of 4

sion that results from repeatedly replicating loops. Increasing the code size may have a negative impact on the instruction cache as well as on the ability of compilers to optimize the code. To manage these concerns, the original compiler algorithm records how large a loop is growing and suppresses transformations that replicate the loop body (e.g. peeling) when it becomes too large. In our experiments however, we have found that the benefits of successfully prefetching needed pages from disk far outweigh the harmful effects of code expansion.

3.2.2 Software Pipelining

For prefetches to be effective, they should be issued early enough that the data arrives in memory before it is needed, but not so early that it risks being replaced before it is used. To hide the latency of disk accesses, we overlap prefetches for a future iteration with the computation of the current iteration using a technique called *software pipelining*. A simple example of this technique is shown in Figure 3.9. The important part of the software pipeline is the *steady state*, where we have both prefetches issued and computation performed. The *prolog* section is used to initialize the pipeline, while the *epilog* performs

(a) Code Before Strip Mining

```
for (i = 0; i < n; i++) {
  if ((i mod 64) == 0)
    f(i);
  g(i);
  if ((i mod 64) == 0)
    h(i);
}
```

(b) Code After Strip Mining

```
for (j = 0; j < n; j+=64) {
  f(j);
  for (i = j; i < j+64; i++)
    g(i);
  h(j);
}
```

Figure 3.8: Example of strip mining a loop by a factor of 64.

the last few iterations where prefetching is no longer needed.

Given that loop iterations are used as the unit of scheduling in the pipelining algorithm, two key issues need to be resolved to schedule prefetches effectively. The first issue is unique to I/O prefetching: how to choose the proper loop across which to pipeline in the presence of multiple loop nests. As discussed in section 3.2.1 with regard to strip-mining, it is possible that several levels of loop nests access less than a page of data. Not only is it impossible to strip-mine these loops, attempting to software pipeline across them is also ineffective since the pipeline never gets into the steady state. The second issue, common to both cache and I/O prefetching, is determining the *prefetch distance* (i.e. the number of iterations in advance that prefetches should be issued).

To illustrate how the proper choice of pipeline loop is affected by the presence of small loop bounds, it is useful to first look at an example of how software pipelining is used to transform code and schedule prefetches. We will thus tackle the problem of finding the prefetch distance first, and examine a simple example of how prefetches are scheduled. We will then consider a slightly more complicated example, where the correct choice of pipeline loop is essential to scheduling the prefetches effectively.

(a) Original Loop

```
for (i = 0; i < 100000; i++)
  A[i] = 0;
```

(b) Software Pipelined Loop

```
prefetch_block(&A[0], 24);                      /* Prolog */

for (i1 = 0; i1 < 88064; i1 += 2048) {          /* Steady State */
  prefetch_block(&A[i1+12288], 4);
  for (i = i1; i < i1 + 2048; i++) {            /* Strip-mined loop */
    A[i] = 0;
  }
  release_block(&A[i1], 4);
}
for (i = 88064; i < 100000; i++)                /* Epilog */
  A[i] = 0;
release_block(&A[88064], 24);
```

Figure 3.9: Example of how software pipelining is used to schedule prefetches the proper amount of time in advance. For this example, 12,288 iterations are required to hide I/O latency.

Finding the Prefetch Distance

Given the amount of latency that needs to be hidden, the problem of finding the prefetch distance (in terms of the number of iterations of the pipeline loop) is simply a matter of determining how much computation is needed. If the latency is expressed as a number of cycles, and we assume that each instruction takes a single cycle, then the number of iterations required, d is given by:

$$d = \left\lceil \frac{l_m}{s + l_p} \right\rceil \tag{3.3}$$

where d is the prefetch distance, l_m is the expected I/O latency, s is the number of instructions in the shortest path through the loop body, and l_p is the software overhead introduced by adding a prefetch instruction to the loop body. In general, it is impossible to determine exactly how many instructions will be executed, however we choose the shortest path to ensure that prefetches are issued early enough. The prefetch overhead latency parameter, l_p, is used to more closely approximate the time spent executing a loop iteration *after* a prefetch request is inserted in the loop. This consideration is especially important for short loop bodies since the time spent issuing the prefetch can be a significant fraction of the time to execute an iteration. The ceiling of the ratio is used to ensure that all of the latency is hidden.

Figure 3.9 shows a simple example of how prefetches and releases are scheduled using software pipelining. For this example, we have chosen parameters that correspond to our experimental architecture: the page size is 4096 bytes, the I/O latency is 100,000 cycles, and the prefetch latency is 10,000 cycles. Since the A[i] reference has spatial locality, the `i` loop is first strip-mined into loops `i1` and `i` using a block size of four pages. Using equation 3.3, the compiler then determines that six iterations of the outer `i1` loop (with a prefetch call added) are needed to fully hide the I/O latency. To initialize the pipeline, a *prolog* is constructed to prefetch the first 24 pages in a single block prefetch call, thus minimizing system call overhead. This is in contrast to cache prefetching where a prolog *loop* would have been constructed to request each page independently. Next, the *steady state* loop is executed. In this loop, blocks of four pages are prefetched in every iteration of the `i1` loop, and used in the inner `i` loop. After each block of pages has been used, block release calls are issued to free the pages. Finally, the *epilog* loop performs the final iterations of computation, after which a block release call frees the last block of pages. This example presents a somewhat idealized version of how release requests are scheduled. In practice, we are also concerned about reducing the number of system calls, hence release operations are bundled with prefetches in a single call whenever possible. When there is no prefetch with which to bundle the release, we simply suppress issuing it.

Now that we have seen how software pipelining schedules prefetches using prolog, steady state, and epilog sections, we return to the question of how to select the right loop across which to pipeline.

Choosing the Pipelining Loop

Ordinarily, prefetches are software pipelined around the innermost loop nest that changes the value of the array indexing function (i.e. the first loop that changes the address referenced). For example, in Figure 3.10(a) the `m` loop would be chosen as the pipeline loop. Also, since the `m` loop has spatial locality, we would like to request blocks of

(a) Sample Code Without Prefetching

```
for (i = 0; i < 64; i++) {
  for (j = 0; j < 64; j++)
    for (k = 0; k < 64; k++)
      for (l = 0; l < 5; l++)
        for (m = 0; m < 5; m++)
          A[i][j][k][l][m] = 0;
}
```

(b) Code After Software Pipelining (original)

```
for (i = 0; i < 64; i++) {
  for (j = 0; j < 64; j++)
    for (k = 0; k < 32; k++)
      for (l = 0; l < 5; l++) {
        prefetch_block(&A[i][j][k][l][0], 12);     /* Prolog */
        for (m = 0; m < 5; m++)
          A[i][j][k][l][m] = 0;
      }
}
```

(c) Code After Software Pipelining (new)

```
for (i = 0; i < 64; i++) {

  prefetch_block(&A[i][0][0][0][0], 12);         /* Prolog */

  for (j0 = 0; j0 < 45; j0 += 5) {               /* Steady State */
    prefetch_block(&A[i][j+19][0][0][0], 4);
    for (j = j0; j < j0 + 5; j++)
      for (k = 0; k < 32; k++)
        for (l = 0; l < 5; l++)
          for (m = 0; m < 5; m++)
            A[i][j][k][l][m] = 0;
  }

  for (j = 45; j < 64; j++)                      /* Epilog */
    for (k = 0; k < 32; k++)
      for (l = 0; l < 5; l++)
        for (m = 0; m < 5; m++)
          A[i][j][k][l][m] = 0;

}
```

Figure 3.10: Example of Software Pipelining with Small Loop Bounds.

four pages with each prefetch. Since each iteration of the `m` loop accesses only 4 bytes of data, it would take 4096 `m` loop iterations to cover a four page block, which means that strip-mining is not performed since only five iterations are available. Using equation 3.3, the compiler determines that prefetches need to be issued roughly 12,000 `m` loop iterations ahead, causing a block prefetch for 12 pages to be inserted for the prolog section of the pipeline, as shown in Figure 3.10(b). Now, since the spatial locality of the `A[i][j][k][l][m]` reference suggests that we only need to prefetch every 4096 `m` loop iterations, and there are only five iterations in the code, the steady state and epilog sections of the pipeline are not created. The effect is that when new pages are prefetched at all, the request is issued just before entering the `m` loop as part of the prolog, but are never early enough to be useful. The fundamental problem is that the pipeline never gets into the steady state.

To cope with this problem, we modified the scheduling part of the algorithm to consider the amount of data actually used in what would ordinarily be chosen as the pipeline loop. If the pipeline loop has spatial locality, and the total data traffic across *all* iterations of that loop is less than a block (i.e. four pages in our experiments), then we choose the next surrounding loop nest as the pipeline loop instead. We apply this heuristic recursively until a loop that accesses more than a block of data, or the outermost loop is found. The result of this modification is shown in Figure 3.10(c), where prefetches are software pipelined across the `j` loop, rather than the `m` loop. It is now possible to schedule prefetches early enough to hide all the latency.

Scheduling Indirect Prefetches

Since indirect references such as `A[index[i]]` are assumed to have no locality, it is not necessary to perform any loop splitting transformations (i.e. the prefetch predicate is always "True"). Indirect prefetches are scheduled using software pipelining in the same manner as direct prefetches, with one minor modification. In addition to fetching the indirect `A[index[i]]` reference, it may also be necessary to schedule a prefetch for the indexing reference, `index[i]`. The indexing reference is treated like any other reference for the purposes of determining locality, however for scheduling, it needs to be prefetched early enough to be used in the *prefetch* of the indirect reference, rather than in the indirect reference itself. More details on prefetching indirect references are given in Mowry's thesis on cache prefetching [21].

A More Detailed Example

Figure 3.11 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]` references). Notice 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]`). Similarly, we convert the prolog loops from the original algorithm into block prefetches whenever possible, as shown in the first two lines of Figure 3.11(b). For references without spatial locality—e.g., `a[b[i]]`—we prefetch only a single page at a time. Also notice how the `b[i]` reference is prefetched well in advance of the prefetch for `a[b[i]]` so that the data will be available to compute the prefetch address. Finally, this example also shows how we bundle prefetch and release requests together whenever appropriate, to minimize the number of system calls.

(a) Original Code

```
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];
```

(b) Code with Prefetching

```
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];
```

Figure 3.11: 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.)

Chapter 4

Evaluation of Proposed System

To evaluate the scheme that we have proposed for automatically tolerating I/O latency, we have constructed an implementation of our ideas and studied its effect on performance for a number of I/O intensive applications. This chapter is organized as follows. We begin in Section 4.1 by presenting our experimental framework, including the hardware on which our system is running, the software infrastructure, and the benchmark applications that we study. Our experimental results are presented in Section 4.2. Section 4.2.1 discusses the overall results in terms of the effect of our approach on execution time. The impact of our approach on system resources is discussed in Section 4.2.2. Section 4.2.3 examines the effectiveness of the compiler and the run-time layer at scheduling prefetches and reducing overhead. In Section 4.2.4 we look at the effect of varying problem sizes on our results. Finally, we present a detailed case study of one of the applications (Buk) in Section 4.2.5.

4.1 Experimental Framework

We now describe our experimental platform including the hardware platform and the software infrastructure used, and the applications which we study in our experiments.

4.1.1 Hardware and Software Infrastructure

The experimental platform used to evaluate our scheme is the Hurricane File System (HFS) [18] and Hurricane operating system [34] running on the Hector shared-memory multiprocessor [35]. Hurricane is a hierarchically clustered, micro-kernel based

Table 4.1: 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

operating system that is mostly POSIX compliant. The HURRICANE micro-kernel provides basic interprocess communication and memory management facilities, including support for mapped file I/O. Most of HFS is implemented outside of the micro-kernel as a user-level server. HFS implements files using *building blocks* to specify the structure of the file and the file system policies applied to a file [18]. Applications are allowed to specify the structure of the file (for instance, the layout of data across the disks) at creation time, and to dynamically change the policies applied when using a file (for example, for replicated files, the application can specify which replica should be used). The basic characteristics of our experimental platform (with the instrumentation disabled) are shown in Table 4.1, and more detailed descriptions of the platform can be found in earlier publications [18, 34, 35]

Our experiments are performed on a 16-processor HECTOR prototype with seven Conner CP3200 disks attached to it. Each disk is directly attached to a different processor and the local processor is responsible for initiating all requests to its disk and servicing all interrupts from its disk. 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. We chose to evaluate our ideas under this system primarily because we had access to the source code for Hurricane and were able to modify it to support the required operations and to provide detailed information on system level activities. The fact that the hardware platform is a multiprocessor is largely irrelevant, the key feature is the disk array that provides our applications with the bandwidth that they require.

Hurricane supports our approach as follows. First, when a prefetch request is received, the kernel checks to see if the specified page is already in memory (or if another read request for that page has already been scheduled). If the page is not in memory, the memory manager allocates a physical page from the free list to hold the file data, an asynchronous read request is sent to the Hurricane file system, and control is returned to the application. The disk scheduler treats prefetches the same as normal disk read requests (both read and prefetch requests are serviced ahead of write requests). In the event that there are no pages on the free list, the memory manager drops the prefetch request and clears the corresponding bit in the shared bit vector to indicate that the requested page has not been fetched. We believe this to be a reasonable strategy since prefetch requests are non-binding performance hints that do not need to be satisfied for program correctness. Also, we want to encourage prefetching applications to balance their memory requirements by explicitly releasing pages that they no longer need—thus, when all memory is in active use it is better to drop the prefetch than risk replacing data that will be needed before the prefetched data. Second, when a release request is received, the kernel removes the mapping for that page from the process's page table (and from the TLB if necessary) and places the page at the end of the free list. We choose to place explicitly released pages at the end of the free list rather than at the head so that they can be reclaimed easily if imperfect compiler analysis causes the page to be released too early. Third, the operating system supports the shared bit vector by allocating a physical page for a process that requests this service. The address of the bit vector is stored in the process's address space descriptor; whenever a page fault occurs, or pages are reclaimed the kernel checks the address space descriptors of the affected processes and sets or clears bits appropriately if a bit vector has been allocated. In our system the runtime layer is responsible for determining the granularity of the bit vector and informing the kernel.

In addition to adding prefetch and release operations to HURRICANE and supporting the shared page, we also added extensive instrumentation to enable us to observe the effect of each static prefetch request. Each static prefetch instruction in the code is given a unique identifier by the compiler. This identifier is passed to the kernel together with the prefetch address and the number of pages to fetch. When a prefetch request is received by the kernel, we record the requested address, the time the request was received, and the identifier of the static prefetch instruction in a *prefetch record*. When a page fault occurs, the kernel checks if the faulting page was prefetched, and updates a set of counters based on the result. This technique allows us to observe the success rate of each static prefetch instruction added to the code by the compiler, and was essential for us to be able to identify the parts of the compiler algorithm that needed to be modified to handle I/O prefetching efficiently. This instrumentation is also used to produce the detailed statistics shown in subsequent sections.

We implemented our prefetching algorithm as a pass in the SUIF (Stanford University Intermediate Format) compiler [32]. The output from the SUIF compiler is C code containing prefetch and release calls (as illustrated earlier in Figure 3.2(b)). We also use the SUIF compiler to convert the original Fortran source code of each application into C code for the original, non-prefetching versions that we use in our experiments. We then compile the resulting C code into a HURRICANE executable using gcc version 2.5.8 with the -O2 optimization flag. During the second compilation step, the prefetching versions are linked to a set of library routines that implement the run-time layer support. Routines to check and set the bit vector to filter prefetch requests are inlined for performance.

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

Table 4.2: 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
CCM	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
FFT	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

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[1]; 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.

4.1.2 Benchmark 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.

A brief description of each of the benchmarks, the data set used, and the execution time of the original non-prefetching versions on our experimental platform is given in

[1]In particular, the shared bit vector is accessed by both the kernel and the application. Hence, the frequent reads by the run-time layer must be uncached, adding to the overhead of filtering prefetch requests

Table 4.2. BUK sorts a large array of integers using the *bucket sort* algorithm and contains both direct and indirect references. This program illustrates a number of features of I/O prefetching, yet is easy to understand, and is thus the subject of the case study in Section 4.2.5. CGM is an example of a sparse-matrix computation; it also contains both direct and indirect references. EMBAR has an extremely simple data access pattern—it repeatedly references a single large array to perform a monte-carlo simulation. MGRID, in contrast, has a very interesting access pattern, despite all the references being direct. In this application, a wavefront moves through a uniform three-dimensional grid representing a potential field. At each point, a computation is performed using the center of the wavefront and the 27 nearest neighboring points (i.e. 6 points that differ by one in only one index, 12 points that differ by one in exactly two of the indices, and 8 points that differ by one in all three indices) [2]. Although the references are all regular, and the pattern is detectable by the compiler, it would be extremely difficult to identify the pattern dynamically in the operating system. FFT solves three-dimensional partial differential equations using both forward and inverse fast-Fourier transforms. The data set is accessed in a different order as this application switches between forward and inverse FFT phases. Finally, APPLU, APPSP and APPBT all solve systems of coupled partial differential equations, using different methods. The structure of these three applications is similar, although they manage their data in slightly different fashions. The common characteristic that is important in our experiments is that they all use multi-dimensional loops in which the innermost dimensions are very small.

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 input 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. For our primary experiments, we target a problem size that uses approximately twice the available memory in our system.

4.2 Experimental Results

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

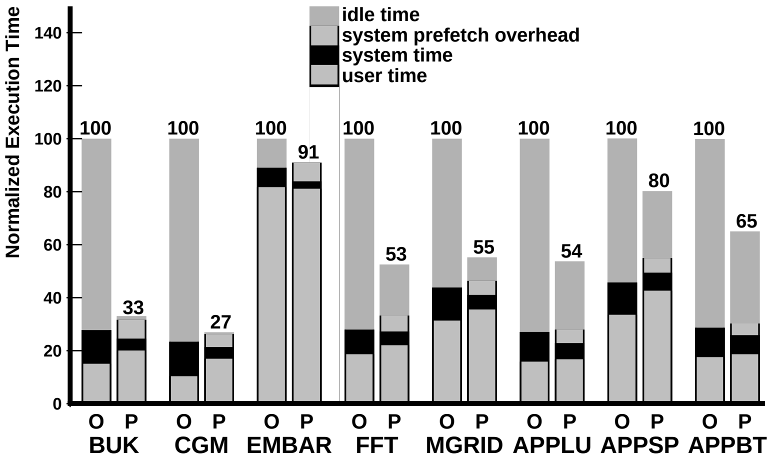
4.2.1 Performance Improvement

Figure 4.1(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 prefetch and release data explicitly (**P**). In each bar, the top section is the amount of time when the processor 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 operating system performing prefetch operations.

As we see in Figure 4.1(a), the speedup in overall performance ranges from 9% to 270%, with the majority of applications speeding up by more than 80%. Figure 4.1(b) presents additional information on page faults[2] and stall time. As we see in Figure 4.1(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 4.1(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% in-

[2]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) Execution Time Breakdown (**O** = original, **P** = with prefetch)

Benchmark	Original: Total Faults (x1000)	Original: Avg. Stall Time (msec)	With Prefetch: Total Faults (x1000)	With Prefetch: Avg. Stall Time (msec)	With Prefetch: 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%
FFT	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 4.1: Overall performance improvement from prefetching

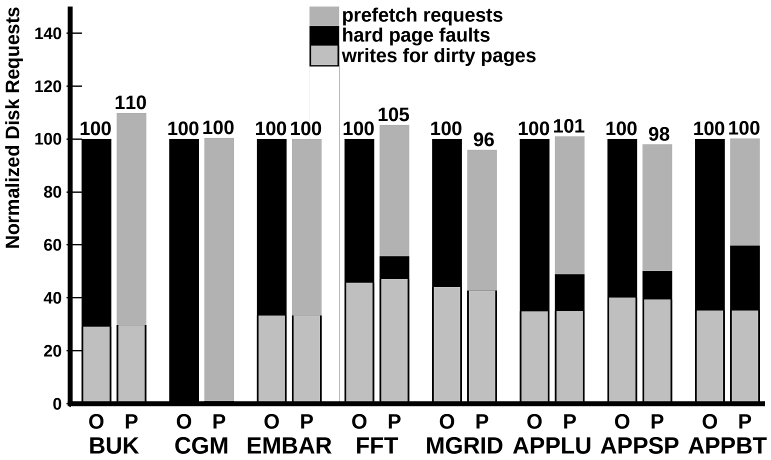
crease in user time in five of the eight applications—in the worst case (CGM), the user time increases by 70%. However, in all cases this increase is quite small relative to the reduction in I/O stall time. If we focus on the system-level overhead of performing prefetch operations, we see in Figure 4.1(a) 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 from a high level perspective, we now look at the impact of explicitly prefetching and releasing data on system resources.

4.2.2 Disk and Memory Utilization

In Figure 4.2(a) and (b) 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 from writing out dirty pages that will be referenced again soon. Hence the increased disk utilization shown in Figure 4.2(b) is simply due to the fact that we are performing roughly the same number of disk accesses over a shorter period of time. Although we have increased disk utilization by a considerable amount, the disks are still idle more than half of the time.

Memory usage during each application's execution is summarized in Figure 4.2(c). 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. This result occurs because these applications return any pages that are not actively being used to the system and the working set is significantly smaller than the total data set. We expect that being able to increase the amount of free memory in the system would greatly reduce the impact of an out-of-core program



(a) Breakdown of requests sent to disk (**O** = original program, **P** = with prefetch)

Benchmark	Original	With Prefetch
BUK	11.8%	40.1%
CGM	11.6%	46.0%
EMBAR	5.9%	9.0%
FFT	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

	Original			With Prefetch and Release			
Benchmark	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%
FFT	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%

(c) Memory sub-system activity and amount of free memory

Figure 4.2: Impact of prefetch and release on system resources

on other applications in a multiprogrammed environment, and we intend to explore this issue further in future work.

4.2.3 Effectiveness of the Compiler and Run-Time Layer

Figure 4.3 presents 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. We begin by examining the success of the static analysis performed at compile-time and then look at how well the run-time layer adapts to the dynamic conditions during execution.

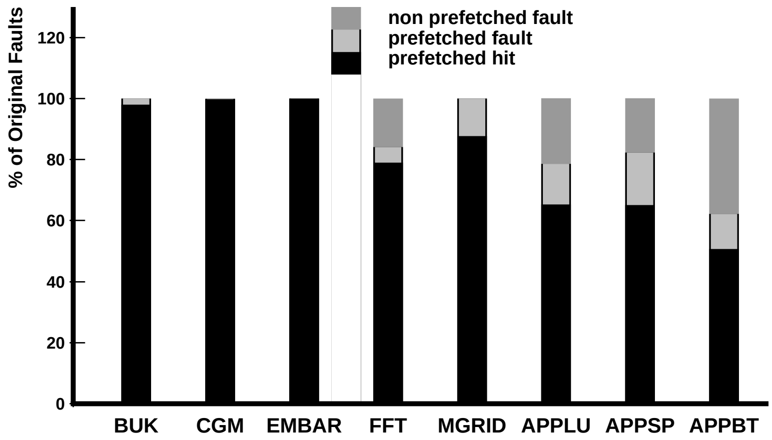
The Compiler

To assess the compiler analysis, we consider what happens to the original page faults when prefetching is added. There are three possibilities: (i) a previously faulting page is successfully prefetched (we call this a *prefetched hit*), (ii) a faulting page is prefetched but still faults when the reference occurs (we call this a *prefetched fault*), and (iii) no prefetch is issued for a faulting page (this is a *non-prefetched fault*). We refer to the combination of the first two cases as the *coverage factor* (i.e. the fraction of original page faults that were prefetched). Figure 4.3(a) shows a breakdown of the impact of prefetching on the original page faults in each application that we study. In all cases except APPBT, the coverage factor is greater than 75% (in four cases, it is greater than 99%), indicating that the compiler is quite successful at identifying references that need to be prefetched. In the half the cases, however, we see that there are still a non-trivial number of page faults that are not prefetched.

Although in most cases the coverage is extremely good, we are interested in discovering why the compiler sometimes fails to prefetch needed data. There are three basic causes that can contribute to poor coverage for the types of applications that we are interested in[3]. Briefly stated, they are:

1. Loop bounds and array dimensions may be unknown at compile time.

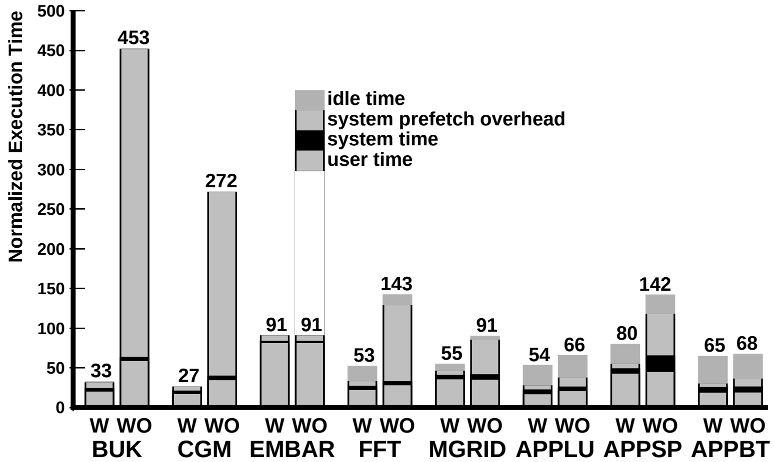
[3]A fourth cause involves data that are not accessed via affine array references (eg. pointer-based codes) but these types of programs are beyond the scope of what we are trying to deal with.



(a) Coverage Factor

Benchmark	Unnecessary Prefetches Issued to OS	Inserted Prefetches Filtered at Run-Time	Total Prefetches Considered at Run-Time
Buk	0.07%	99.79%	25,216,058
Cgm	0.08%	99.74%	53,285,582
Embar	0.00%	0.02%	65,537
Fft	7.99%	99.59%	39,874,709
Mgrid	8.03%	99.17%	9,004,863
Applu	3.75%	96.99%	2,529,757
Appsp	7.55%	99.51%	89,813,618
Appbt	2.54%	98.31%	4,008,500

(b) Unnecessary Prefetches



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

Figure 4.3: Effectiveness of the compiler analysis and run-time filtering.

Example Code

```
for (i = 0; i < 100; i++) {
  foo(&A[i]);          /* not recognized as an array reference  */
  bar(B[i]);
}
```

Figure 4.4: Example of a reference not identified by the compiler

2. Data may not be aligned well with respect to page boundaries.

3. Some references that should be prefetched do not look like array accesses when the prefetching pass of the compiler is executed.

Each of these causes are a contributing factor in the relatively poor coverage for Applu, Appsp and Appbt.

First, when both the loop bounds and the array dimensions are unknown, the compiler must make an assumption about the amount of data accessed in the loop. If the compiler incorrectly assumes unknown loops to be small, it can fail to schedule needed prefetches. The fundamental problem is that such loops appear to be localized, implying that reuse can be exploited and prefetching is needed only for the first reference. Unfortunately, simply assuming unknown bounds to be large is not a reasonable solution since it can cause another scheduling problem if the loops are actually small (as discussed in Section 3.2.2). A better solution would be to have the compiler generate multiple versions of the loop. At run-time the correct version to execute could be chosen based on the actual values of the loop bounds. The question of how to handle quantities that are unknown at compile time is a subject for further investigation.

The second instance in which some pages are not prefetched arises when the data is not well aligned with respect to the page boundaries. As mentioned in Section 3.1.3, the compiler assumes that two references will probably touch the same page if the data locations that they access are separated by less than half a page. If the two data locations are actually on *adjacent* virtual pages, then only the first page will be prefetched and the second will still suffer a page fault. In general we expected this problem to be rare, however, it arises relatively frequently in Applu, Appsp, and Appbt.

The final problem is best explained with the use of an illustrative example. Consider the code shown in Figure 4.4. In this loop, procedure `foo` is called with *the address* of `A[i]`, while procedure `bar` is called with *the value* of `B[i]`. Ideally, we would like to prefetch both of these references, but the current prefetching pass of the compiler is only able to recognize the `B[i]` reference. The reason is that earlier passes of the SUIF compiler have converted the high-level source code shown in Figure 4.4 to an internal representation where `&A[i]` is not identified as an array reference[4]. This situation arises during the initialization phase in APPLU, APPSP, and APPBT. While this problem does not contribute greatly to the non-prefetched page faults in these applications, it could be an important case to recognize in general.

Having shown why the compiler is unable to prefetch all of the original faults, we now turn our attention to the *prefetched fault* category in Figure 4.3(a). A page fault can occur for prefetched pages for two reasons: either the prefetch has not had time to complete and the page has not yet arrived in memory, or the prefetch was issued far too early and the page has been replaced from memory. This category reflects the effectiveness of our compiler in scheduling prefetches the right amount of time in advance. In the cases where prefetched faults are noticeable in Figure 4.3(a), the problem is almost always that the prefetches were not issued early enough.

Two observations help to explain why prefetches may not have time to complete before the data is needed. First, the compiler schedules prefetches so that they will be issued early enough during the *steady state* of the software pipeline. Prefetches issued in the prolog sections are intended to initialize the pipeline, but are not scheduled to complete before the data is needed. Second, when loop bounds in multi-dimensional loops are unknown at compile-time, the compiler may pipeline around the wrong loop nest. As shown in Section 3.2.2, the result is that prefetches are only issued in the prolog and the steady state is never reached. Again, we expect that to correct this problem we will need to generate multiple versions of such loops and choose the right one to execute

[4]In fact, handling this case properly may require interprocedural analysis, since whether or not the data at `&A[i]` needs to be brought into memory depends on what the procedure `foo` does with the argument that is passed to it. Also, scheduling a prefetch properly for this data depends on *when* it is used by `foo`.

dynamically when the program is executed.

Finally, the middle column of Figure 4.3(b) shows that most of the prefetch requests scheduled by the compiler are actually *unnecessary* (i.e. the page was already mapped into memory) and are filtered out by the run-time library. For BUK and CGM, most of these unnecessary prefetches result from always prefetching indirect references. The compiler assumes that each such reference could touch a different page of data, but for these applications this "worst-case" behavior rarely occurs. We will examine the utility of prefetching indirect references more closely during our case-study of BUK in Section 4.2.5. In all cases, unnecessary prefetches occur whenever the compiler underestimates memory's ability to retain data. One cause of this effect is not considering independent loops—if two independent loops access the same data, both will be treated as the first reference to that data, regardless of the amount of data accessed. In general, this problem is extremely difficult to solve since it may require interprocedural analysis to evaluate all the loops in a program. A notable exception to the problem of unnecessary prefetches is EMBAR; the data access pattern in this program is simple enough to be analyzed perfectly by the compiler.

The Run-Time Layer

The primary purpose of the run-time layer is to efficiently filter out prefetches for pages that are already in memory, thus reducing the overhead of the unnecessary prefetches scheduled by the compiler. To evaluate the effectiveness of the run-time layer at this task, Figure 4.3(b) presents statistics on how many prefetches were *unnecessary*. 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 4.3(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 middle column of Figure 4.3(b) shows the fraction of dynamic prefetches that were inserted by the compiler and filtered out by the run-time layer. As discussed in the previous section, it would be extremely difficult to remove many of these unnecessary prefetches statically, making run-time filtering the best option for reducing overhead. Finally in the third column of Figure 4.3(b) we show how many prefetches are checked by the run-time layer. The large numbers in this column provide a strong argument for making the filtering as efficient as possible, and also help to explain why the user time component in Figure 4.1(a) increases significantly for some applications.

Figure 4.3(c) quantifies the performance advantage of the run-time layer. As we see in this figure, 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 operating system. From these results, we conclude that the run-time layer is clearly an essential component for achieving good performance in our system.

4.2.4 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.

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 4.5 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

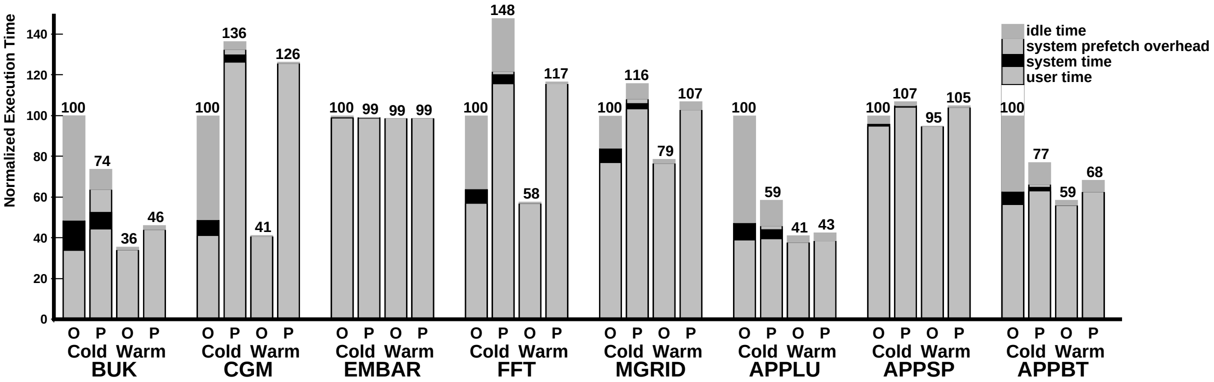


Figure 4.5: Performance with in-core data sets (**O** = original, **P** = with prefetch; **Cold** = cold-started, **Warm** = warm-started). Performance is normalized to the original, cold-started cases.

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 address 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.

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 4.6 shows the performance of four applications where the problem size is 4-10 times larger than the available memory.

For FFT, APPLU and APPBT the problem size used in this experiment is approximately 200 MB, which requires that each bit in the bit vector represent two contiguous virtual memory pages (Recall from Section 2.4 that we restrict the size of the bit vector to a single page). A larger size is used for MGRID because the structure of the program requires that the data set be cubical. The problem size used in our earlier experiments was only only 20% larger than the available memory–the next larger problem size (shown in Figure 4.6) requires 464 MB of memory, which is approximately 10 times more than

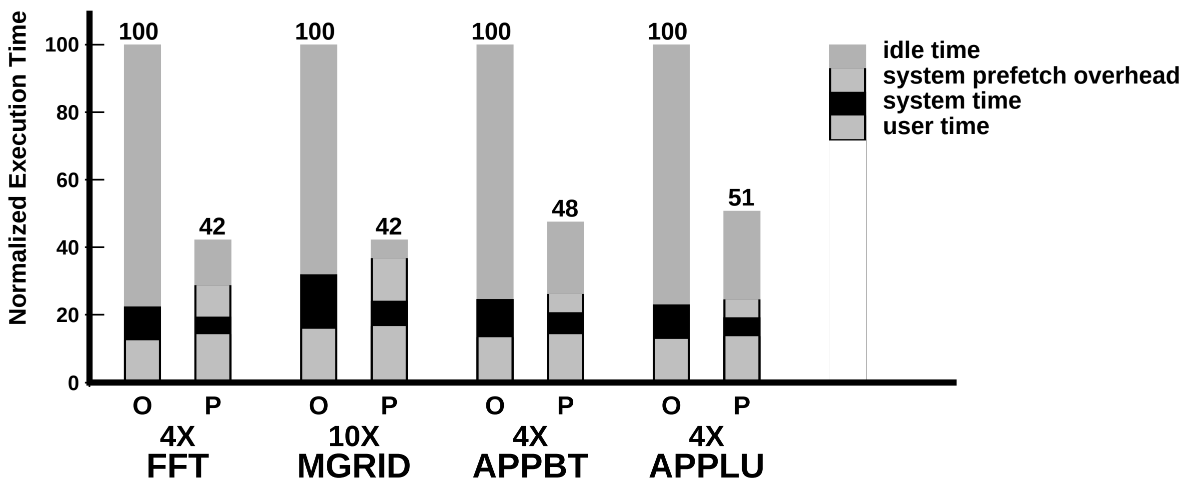


Figure 4.6: Performance with larger out-of-core problem sizes. Numbers above application names indicate how much larger the problem sizes are than available memory.

what is available, hence each bit in the bit vector represents four pages.

The granularity of the bit vector can potentially have an impact on performance because the run-time layer is given a less detailed view of the state of main memory. The results in Figure 4.6, however, show that for these applications the performance improvements remain large. In fact, prefetching offers slightly larger speedup in all these cases since there is more I/O latency to hide. In addition, Applu and Appbt benefit from the coarser granularity since fetching both pages in the set corresponding to a given bit automatically solves the alignment problem discussed in Section 4.2.3.

4.2.5 Case Study: Buk

In this section we take a closer look at how explicitly prefetching and releasing pages affects application performance by focusing on a single application. We have chosen to examine Buk for this case study for three reasons: (i) the program is short and easy to understand; (ii) the problem size can be scaled linearly; and (iii) the program contains both direct and indirect references, allowing us to evaluate the costs and benefits of indirect prefetches. We begin by describing the computation and data accesses performed in Buk before looking at the indirect prefetches. Finally, we examine what happens to execution time as we move from in-core problem sizes to out-of-core problem sizes, both with and without prefetching.

Description of Application

Figure 4.7 shows the main computations performed by BUK as C source code[5]. BUK takes an array of unsorted integers that are held in `key`, computes the the position that each integer should have when sorted and stores the position in `rank`, and finally copies the integers from `key` into `key2` using the values stored in `rank`.

The computations performed in BUK are organized in two phases. During the `bucksort` procedure, the input array `key` and the temporary storage array `rank` are both accessed using direct references. The temporary array `keyden` is used to record the number of times each distinct value in the input array `key` occurs, and then to calculate the position each integer should have when sorted. The `keyden` array is accessed by direct references in some loops, and by indirect references in others. We refer to this phase as the *ranking phase* in our discussion. The second phase of computation occurs after `bucksort` in the `main` procedure when the integers are sorted by copying each one from `key` to its proper position in `key2` using the values stored in `rank`. In this phase, `rank` and `key` are accessed by direct references, while `key2` is accessed indirectly. We will refer to the second phase as the *copying phase* in our discussion.

All of the array references in BUK are identified and prefetched by our compiler (the bar for BUK in Figure 4.3(a) shows that the coverage factor is 100%).

Benefits of Indirect Prefetches

Since the compiler has no information about the locality of indirect references a decision must be made at compile time to either always or never prefetch them. To evaluate the costs and benefits of prefetching indirect references we examine the performance of BUK when only direct references are prefetched, and when both direct and indirect references are prefetched. In addition to the overall effects, we also consider the impact of prefetching indirect references in each phase of the program separately. The results of these experiments are shown in Figure 4.8. In this figure, all bars have been normalized to

[5]The actual program that we use in our experiments is written in FORTRAN. In our C representation of this code, we only show the computations that are relevant to this discussion

```
extern int main() {
  int i, j;
  int key[8388777], key2[8388777], rank[8388777], keyden[524288];

  /* Get the rank for each key */
  bucksort(key, rank, keyden, 8388608, 524288);

  /* Copy keys in sorted order into key2 */
  for (i = 0; i <= 8388607; i++)
    key2[rank[i]] = key[i];

  /* Test if any keys are out of order */
  j = 0;
  for (i = 0; i <= 8388606; i++) {
    if (key2[i + 1] < key2[i])
      j = j + 1;
  }
  if (j == 0) printf("PASSED: 0 out of place.");
  else printf("FAILED: %d out of place", j);

  return 0;
}

extern int bucksort(int *key, int *rank, int *keyden, int n, int maxkey) {
  int i;

  /* Zero the keyden array */
  for (i = 0; i < maxkey; i++)
    keyden[i] = 0;

  /* Count occurrences of each key (the 'key density') */
  for (i = 0; i < n; i++)
    keyden[key[i]] = keyden[key[i]] + 1;

  /* Create running sum (i.e. starting index) of keyden array */
  for (i = 1; i < maxkey; i++)
    keyden[i] = keyden[i] + keyden[i - 1];

  /* Compute rank for each key */
  for (i = 0; i < n; i++) {
    keyden[key[i]] = keyden[key[i]] - 1;
    rank[i] = keyden[key[i]];
  }
  return 0;
}
```

Figure 4.7: Source code (C representation) for BUK

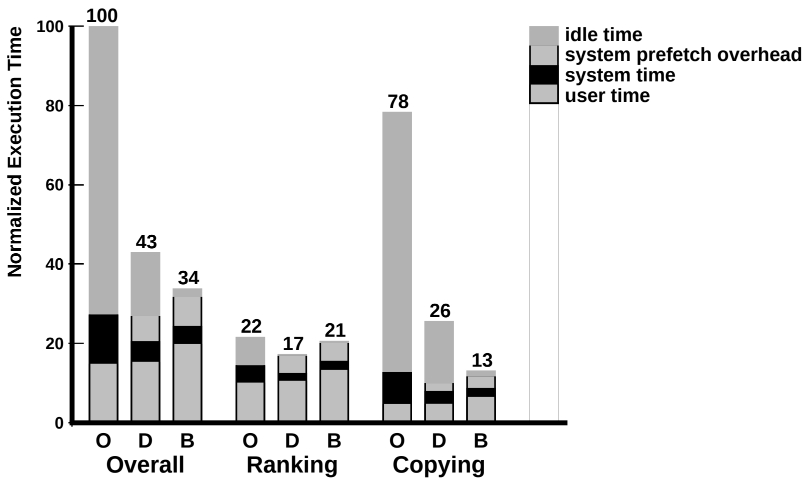


Figure 4.8: Performance for different phases of Buk. (**O** = original, **D** = with direct-only prefetching, **B** = with direct and indirect prefetching). Bars are normalized to the original, non-prefetching case.

the original, non-prefetching execution time. The first set of three bars (labelled **Overall**) show the execution time breakdown for the entire program for the original (**O**), direct-only prefetching (**D**), and both direct and indirect prefetching (**B**) versions. From these bars, it can be seen that prefetching indirect references reduces execution time by an additional 9% over direct prefetching alone. The second set of three bars shows what happens during the ranking phase, while the final set of three bars shows what happens during the copying phase of the program. During ranking, the indirect prefetches are for the `keyden` array, which is small compared to the size of memory (only 2 MB) and is always found in core. In this phase, direct prefetching alone is able to capture all the important references and the indirect prefetches merely introduce overhead with no benefit. In the copying phase however, the indirect prefetches are for the `key2` array, which is much larger (33 MB). During this phase, a large number of important references are missed by only prefetching the direct references. The additional overhead of filtering the indirect prefetches is amply offset by the reduction in I/O stall time for the copying phase.

Ideally, we would like to be able to achieve the best of both worlds—avoid scheduling indirect prefetches when the locality of the indirect references is good (as in the ranking

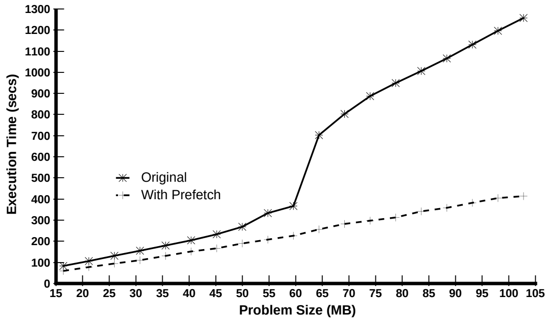


Figure 4.9: Performance of Buk (cold-started) across a range of problem sizes.

phase) and issue indirect prefetches aggressively when the locality is poor (as in the copying phase). Once again, this could potentially be accomplished by creating multiple versions of the loops (one in which indirections are prefetched and one in which they are ignored) and choosing the best one to execute based on the dynamic conditions at run-time.

Crossing the In-core / Out-of-core Boundary

In Buk, the amount of work to be done grows linearly with the problem size. Ignoring page faults, we would normally expect the execution time to also increase linearly with the problem size. To show how performance is affected when an application runs out of physical memory, we executed Buk with problem sizes ranging from roughly one quarter to twice the size of main memory both with and without prefetching. The execution times for each problem size are plotted in Figure 4.9.

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

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

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.

Chapter 5

Related Work

The problem that we address in this thesis has been considered by many other researchers in a range of different areas. We can classify related research into six broad areas: (i) using prefetching to improve the performance of paged virtual memory systems; (ii) using knowledge of application-specific access information to improve the operating system's replacement policies; (iii) determining the best strategy for combined prefetching and caching given complete and accurate information about all future accesses; (iv) prefetching in file systems by automatically detecting access patterns; (v) prefetching in file systems based on information supplied by the application; and (vi) compilation techniques for out-of-core programs. We now look at each of these areas in turn.

The idea of using a compiler to extract access patterns from an application and passing this information to the operating system to improve virtual memory performance is not a new one. The first study in this area was conducted nearly twenty years ago by Trivedi [33], who looked at the use of application access patterns extracted by a compiler to implement "prepaging". Although the interface between the compiler and the operating system is nearly identical to that which we propose, 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 [22]. Thus, our approach is much more widely applicable. Second, we introduce the idea of a run-time layer to filter the prefetches inserted by the compiler, thus allowing the compiler to be much more aggressive about adding prefetches when the analysis cannot be performed perfectly. This component of our system is essential to achieving good performance. Without a way to filter prefetches efficiently, the compiler must be much more careful and may fail to fetch many important references. Earlier work did not try to prefetch indirect references or references in loops where the bounds were unknown.

Other work has also been done in the area of prefetching for paged virtual memory systems, however this research generally depends on the operating system being able to detect patterns to initiate prefetching. Curewitz, Krishnan, and Vitter investigate using techniques developed for data compression to predict what to prefetch [8]. Their target environments are object-oriented databases and hypertext system, and they assume that the CPU will be mostly idle, allowing the operating system to perform complicated calculations without affecting a user's perceived response time. Song and Cho use the fault history to detect patterns and initiate prefetching [28]. These techniques all 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. For instance, in FFT these schemes would incur page faults whenever the application moves between forward and inverse FFT phases.

Using application-specific knowledge to assist memory management replacement policies was studied by Malkawi and Patel [20], and by Park, Scott, and Sechrest [24], however, these schemes only consider retaining needed pages in memory and do not attempt to prefetch. Although the amount of I/O needed can be reduced by intelligent memory management, the latency of the remaining I/O operations is still a serious concern.

Finding the best combined prefetching and caching strategy (i.e. the one that gives the shortest execution time) for a fully known sequence of accesses has been studied by Cao *et al.* [4] for the case of a single disk and by Kimbrel *et al.* [15] for varying numbers of disks. Kimbrel *et al.* show that the best behavior depends on the amount of contention of the disks, but that for reasonable data layouts on more than four disks contention is rarely a problem.

Prefetching in file systems by automatically detecting file access patterns has been well studied [1, 11, 12, 13, 16, 17, 19]. Kroeger and Long look at using the compression technique known as *prediction by partial match* to detect access patterns and to decide what to prefetch [19]. Griffioen and Appleton construct a *probability graph* based on prior file system accesses. Both approaches attempt to improve the performance of the overall file system by predicting which *files* are likely to be referenced next when a particular file is opened. In contrast, our focus is on improving performance for out-of-core applications that typically access a small number of very large files. Once again, techniques that depend on being able to detect an access pattern cannot prefetch until the pattern has been established, may do the wrong thing when the pattern changes, and may have difficulty detecting the types of complicated access patterns that can occur in scientific codes.

Another technique implemented in file systems is to support prefetching based on information supplied explicitly by the application [25, 27, 31]. Of these approaches, the TIP system developed by Patterson et al. [25] 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, they expect applications to fully disclose their access patterns at start-up, relying on the operating system to decide what to do with this information. Thus, no concept of time is embedded in the hints and the operating system must be able to determine when the prefetch should be initiated. The resulting operating system support is more complex than ours, and would be even more difficult to provide in a mapped file or virtual memory system where the operating system does not see regular read requests.

Compiling for out-of-core codes tends to focus on three areas. The first area 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, 14, 23, 30]. 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. The third compilation approach is to take programs that already contain explicit I/O calls, move them to an earlier program point, and change them to asynchronous I/O calls instead. We feel that compiler analysis that targets an I/O interface is limited by the alias analysis problem described earlier in Section 2.2.1, and in general cannot be as aggressive as an algorithm that supports non-binding prefetching.

Chapter 6

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.

We have also shown that good performance can only be achieved by adapting to dynamic conditions, since the compiler analysis may be hampered by quantities that are unknown at compile-time. Towards this end, we have implemented a run-time layer that adapts by filtering out unnecessary prefetches at the user-level without performing an expensive system call. In addition, we have shown that generating even more adaptive code would be useful scheduling prefetches the right amount of time in advance.

6.1 Future Work

We view this work as an encouraging first step, and we are currently extending our research in the following directions:

1. We are planning to implement the support we need within a commercial operating system, so that future experiments can be conducted on state-of-the-art commercial systems (and not just research prototypes). We would expect the potential gains from I/O prefetching to be larger on modern systems, since processor speeds have increased so much relative to disk speeds.

2. We are exploring new ways that the compiler and operating system might interact to allow the application to better adapt its dynamic behavior to the available resources, especially in a multiprogrammed environment. For example, the application could compare the amount of memory it expects it will need with the amount the operating system says is available, to determine how to prefetch or whether to prefetch at all (i.e. if the entire data set fits in the available memory, we would suppress prefetching and thus avoid the overhead shown earlier in Section 4.2.4).

3. We intend to investigate how our approach behaves on a multiprocessor, where there is the additional challenge of maximizing I/O locality such that a processor is likely to access a nearby disk in the system (hence minimizing network traffic).

4. We would like to be able to extend the scope of our work beyond numeric applications; although we can already handle some forms of irregular access patterns (e.g., sparse matrix codes), we would also like to handle pointer-based codes and other non-numeric codes.

Bibliography

[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] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[11] 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.

[12] A. S. Grimshaw and Edmond C. Loyot, Jr. ELFS: object-oriented extensible file systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, page 177, December 1991.

[13] 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.

[14] 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.

[15] T. Kimbrel, A. Tomkins, R.H. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. Karlin, and K. Li. A trace-driven comparison of algroithms for parallel prefetching and caching. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 19–34, October 1996.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] Yoonho Park, Ridgway Scott, and Stuart Sechrest. Virtual memory versus file interfaces for large, memory-intensive scientific applications. In *Proceedings of Supercomputing 96*, pages 17–22, November 1996.

[25] 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.

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

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

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

[29] 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.

[30] 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.

[31] 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.

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

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

[34] 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.

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

[36] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[37] 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.