Explicit Compiler-based Memory Management for Out-of-core Applications

Angela Demke Brown

May 2005

CMU-CS-05-140

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Todd Mowry, Chair
M. Satyanarayanan
Garth Gibson
Monica Lam, Stanford University

This research was sponsored by the National Aeronautics and Space Administration (NASA) under grant no. NAG2-1230 and the National Science Foundation (NSF) under grant nos. CCR-0085938 and EEC-8907068. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any sponsoring party or the U.S. Government.

Keywords: I/O prefetching, compiler analysis, virtual memory management, software pipelining, out-of-core applications, dynamic adaptation.

Abstract

For a large class of scientific computing applications, the continuing growth in physical memory capacity cannot be expected to eliminate the need to perform I/O throughout their executions. For these *out-of-core* applications, the large and widening gap between processor performance and disk latency is a major concern. Current operating systems deliver poor performance when an application's working set does not fit in main memory. As a result, programmers who wish to solve these out-of-core problems efficiently are typically faced with the onerous task of rewriting their application to use explicit I/O operations (e.g., read/write). In many cases, the end result is that the size of physical memory determines the size of problem that can be solved.

In this dissertation, we propose and evaluate a fully-automatic technique which liberates the programmer from this task, provides high performance, and requires only minimal changes to current operating systems. In our scheme, the compiler provides the crucial information on future access patterns without burdening the programmer, the operating system supports non-binding *prefetch* and *release* hints for managing I/O in a virtual memory system, and the operating system cooperates with a run-time layer to accelerate performance by adapting to dynamic behavior and minimizing prefetch overhead. This approach maintains the abstraction of unlimited virtual memory for the programmer, gives the compiler the flexibility to aggressively insert prefetches ahead of references, and gives the operating system the flexibility to arbitrate between the competing resource demands of multiple applications.

We implemented our compiler analysis within the SUIF compiler, and used it to target implementations

of our run-time and operating system support on both research and commercial systems (HURRICANE and IRIX 6.5, respectively). Our experimental results show large performance gains for out-of-core scientific applications on both systems: more than 50% of the I/O stall time has been eliminated in most cases, thus translating into overall speedups of roughly twofold in many cases. Our initial experiments motivated a new compiler scheduling algorithm that is capable of tolerating the large and variable latencies that are common for disk accesses, in the presence of multiply-nested loops with unknown bounds. On our current experimental systems, many of our benchmark applications remain I/O bound, however, we show that the new scheduling algorithms are able to substantially improve performance in some cases, reducing execution time by an additional 36% in the best case. We further show that the new algorithms should enable applications to make more effective use of higher-bandwidth disk systems that will be available in the future.

Acknowledgements

Graduate school is not an experience that can be survived alone. Having finally reached the end of it, I find that acknowledging everyone that has helped me get here is nearly as daunting as producing the rest of the thesis. There are, of course, some obvious places to start.

First, I would like to thank my advisor, Todd Mowry, whom I hold responsible both for getting me into Ph.D. studies and for ultimately getting me out. This dissertation would not exist without his guidance, support, and continual encouragement. Over the years, I have been the recipient of valuable advice on all matters academic, and though I have not always followed it, I have grown as a researcher, a teacher, and a person as a result. I also thank my thesis committee, M. Satyanarayanan, Garth Gibson, and Monica Lam for their input and guidance on this dissertation and the research behind it.

The implementations of the ideas in this thesis would not have been possible without the help and guidance of the Hurricane team at the University of Toronto (in particular, Orran Krieger and Ben Gamsa) and the SGI Irix developers (especially Luis Stevens who showed me around the internals of the Memory Management Control Interface).

There are a large number of people who made my stay at Carnegie Mellon a unique and enriching experience. I would like to thank all the folks at the Parallel Data Lab, including those who ran the show and those who kept it running. Thanks also to the STAMPede group for providing feedback on research ideas and debugging many practice talks. For laughter and other diversions, I thank Greg and Nancy Steffan, Chris Colohan, Chris Palmer, Carrie Sparks, Jason Flinn, Ted and Addie Wong, Rob O'Callahan, Dave Maltz, Joan

Digney, Cheryl Gach and Paul Mazaitis.

Is it possible to thank a department? The environment for computer science research at Carnegie Mellon is phenomenal and far more human than one has any reason to expect. I am sure this is due in no small part to the efforts of Sharon Burks, Associate Department Head and de facto den mother. It is also the result of a dedicated and talented team of support staff who keep the organization running smoothly. Special thanks to the folks in facilities who put up with my personal version of Irix and the extra support it required, and to the folks who supplied me with a spare key to my office so many, many times.

I want to thank friends from before graduate school who have stuck with me and encouraged me through it all. Michelle, Nadine, and Siobain deserve a special thanks for not letting me lose touch with the important things. I am also grateful for the support of my parents, my brother and his family and the world's most wonderful in-laws. Finally, I cannot express how much I owe to my husband Joe, who moved to Pittsburgh and provided patience, love, encouragement and supper while I pursued this degree.

There are so many people who have touched my life during these years as a graduate student, and to whom I owe my heartfelt thanks, that there are surely sins of omission herein. For everyone else not mentioned, please forgive my lapses.

Contents

1 Introduction **1**

1.1 Problems with Out-of-Core Applications 3

1.1.1 Virtual Memory Versus Explicit I/O Interfaces 3

1.1.2 Performance Using Virtual Memory 5

1.1.3 Impact on Other Applications 7

1.2 Related Work 8

1.2.1 Prefetching in Virtual Memory Systems 9

1.2.2 Prefetching for Explicit File Accesses 10

1.2.3 Integrating Prefetching and Replacement 11

1.2.4 Compiling Out-of-Core Programs 12

1.2.5 Application-controlled Memory Management 12

1.3 Research Goals 13

1.4 Contributions 13

1.5 Overview of Dissertation 15

2 System Design Alternatives and Evaluation Framework **17**

2.1 Key Requirements 18

2.2 Design Alternatives 22

2.2.1 Potential Sources of Information 22

2.2.2 Options for Sharing Information . . . 25
2.2.3 Options for Communicating Decisions . . . 27
2.3 Actual System Design . . . 30
2.3.1 The Compiler . . . 31
2.3.2 The Run-Time Layer . . . 33
2.3.3 The Operating System . . . 33
2.4 Evaluation Framework . . . 35
2.4.1 Hardware Platforms . . . 35
2.4.2 Research System Infrastructure . . . 35
2.4.3 Commercial System Infrastructure . . . 36
2.4.4 Benchmarks . . . 38

3 Impact on Out-of-core Applications **39**
3.1 Compiler Algorithm . . . 40
3.1.1 Locality Analysis . . . 41
3.1.2 Scheduling Prefetches . . . 49
3.1.3 Compiler Implementation . . . 57
3.2 An Initial Prototype: HURRICANE . . . 58
3.2.1 Operating System and Run-Time Layer Implementation . . . 58
3.2.2 Evaluation of HURRICANE Implementation . . . 62
3.3 Experience with a commercial operating system: IRIX . . . 78
3.3.1 Implementation . . . 78
3.3.2 Evaluation of IRIX Implementation . . . 80
3.4 Lessons and Limitations . . . 84

4 Performance in Multiprogrammed Environments **85**
4.1 Issues with Interactive Applications . . . 86

4.2 Alternatives for Memory Management . . . 88
4.2.1 Global vs. Local Replacement . . . 88
4.2.2 Application-Managed Replacement . . . 89
4.3 Compiler Support . . . 91
4.3.1 Complications with Generating *Release* Requests . . . 91
4.3.2 An Example of Data Reuse and the Effect on Releases . . . 92
4.3.3 Implementation of Compiler Analysis . . . 93
4.4 Implementation of Operating System Support for Release . . . 95
4.4.1 Setting the Memory Limit . . . 96
4.5 The Run-Time Layer Support . . . 97
4.5.1 Implementation Details . . . 99
4.6 Experimental Results . . . 101
4.6.1 Benchmarks . . . 101
4.6.2 Performance of the Out-of-Core Applications . . . 102
4.6.3 Effectiveness of Releases . . . 106
4.6.4 Impact on Interactive Response Time . . . 109
4.7 Summary . . . 111

5 Improving the compiler scheduling algorithm **113**

5.1 Scheduling challenges . . . 114
5.1.1 Variations in Prefetch Distance . . . 114
5.1.2 Problems with nested loops . . . 119
5.2 Developing the Continuous Software Pipelining Algorithm . . . 121
5.2.1 Calculating prefetch distances for multiple loop nests . . . 121
5.2.2 Nested pipelines . . . 122
5.2.3 Merging prologs and epilogs . . . 132

5.2.4 Reducing code expansion 140
5.2.5 Imperfectly nested loops 143
5.3 Evaluation 146
5.4 Chapter Summary 150

6 Conclusions **153**

6.1 Future Work 154
6.2 Final Observations 156

List of Figures

1.1 Virtual memory performance on two systems for in-core and out-of-core problem sizes. . . . 5

1.2 Time to touch 1MB of data for varying *think times* between accesses. 7

2.1 Improving demand paging performance 19

2.2 Information flow between components of our system. 26

2.3 Example illustrating the importance of non-binding prefetches 28

2.4 Steps in the automatic transformation of original application into prefetching/releasing executable. 31

3.1 Data reuse example. 43

3.2 Example of how loop iteration counts affect locality. 45

3.3 Example of how prefetch predicates are constructed. 48

3.4 Example of peeling the first and last iterations of a loop 50

3.5 Example of unrolling and strip-mining a loop by a factor of 64 50

3.6 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. 53

3.7 Example of Software Pipelining with Small Loop Bounds. 55

3.8 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.) 57

3.9 Implementation of prefetching and releasing support on HURRICANE. 59

3.10 Overall performance improvement from prefetching on HURRICANE. 63

3.11 Impact of prefetch and release on system resources on HURRICANE. 65

3.12 Effectiveness of the compiler analysis and run-time filtering. 67

3.13 Example of a reference not recognized as an array reference (inside `foo`) by the compiler. . 69

3.14 Example of reuse not identified by the compiler. 70

3.15 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. 72

3.16 Performance with larger out-of-core problem sizes. Numbers above application names indicate how much larger the problem sizes are than available memory. 73

3.17 Source code (C representation) for BUK 75

3.18 Performance for different phases of BUK, normalized to the original, non-prefetching case (**O** = original, **D** = with direct-only prefetching, **B** = with direct and indirect prefetching). . . 76

3.19 Performance of BUK (cold-started) across a range of problem sizes. 77

3.20 Implementation of prefetching and releasing support on IRIX. 78

3.21 Overall performance improvement from prefetching and releasing on IRIX (**O** = original, **P** = with prefetching and releasing). 81

3.22 Impact of prefetching on the original page faults under IRIX. 83

4.1 Impact of sharing the machine with an out-of-core matrix-vector multiplication (MATVEC) on the response time of an interactive task across a range of sleep times between touching 1 MB of data. 87

4.2 Example source code showing multiple references with different types of reuse, and graphical view of the data accesses during a single iteration of the innermost loop. 92
4.3 Example of the output of the prefetching compiler. Arguments are: (prefetch address, release address, number of 16KB pages, release priority, request identifier) 95
4.4 Handling prefetches and releases at run-time. 98
4.5 Impact of prefetching and releasing on the execution times of the out-of-core applications. (**O** = original, **P** = with prefetching, **R** = with prefetching and releasing, **B** = with prefetching and release buffering) 103
4.6 Soft page faults due to page invalidations. 105
4.7 Breakdown of outcomes for freed pages. 108
4.8 Impact of releasing on interactive response time. 110

5.1 Effect of calculating prefetch distances at run-time. Bars labeled "O" are the original, non prefetching version; bars labeled "S" use a static compile-time latency; bars labeled "D" use a dynamic latency value (equal to the static value) obtained at run-time. 117
5.2 Adding prefetches for two-dimensional array accesses 120
5.3 Algorithm for calculating prefetch distances in nested loops. 122
5.4 Sample code for software pipelining a single loop with multiple loop index offsets 122
5.5 Nested software pipelines for two-dimensional array prefetches 123
5.6 Comparison of nested pipelining vs. original single-loop pipelining for different latency values 127
5.7 Non-sequential data accesses across outer loop: only the first part of each row of matrix is used 128
5.8 Effect of non-sequential data access across outer loop 130
5.9 Continuous software pipelines for two-dimensional array prefetches 131
5.10 Progression from Nested pipelines to Continuous pipelines 133
5.11 SUIF-style abstract syntax tree for simple procedure with loops 134
5.12 Algorithm for building continuous software pipelines in nested loops 137

5.13 Simulated performance of continuous software pipelining under varying latency and data layouts 139
5.14 Wrapped pipeline: only two copies of the original loop body are needed to cover all situations. 141
5.15 Example of imperfect loop nesting structure found in blts subroutine of applu.f 145
5.16 Effect of new scheduling algorithms. Bars labeled “O” are the original, non prefetching version; bars labeled “D” use a dynamic latency value with the original scheduling algorithm; bars labeled “N” use the nested pipelining algorithm; bars labeled “C” use the continuous pipelining algorithm. 149

List of Tables

2.1 Available Information Sources 22

2.2 HECTOR/HURRICANE characteristics 36

2.3 SGI Origin 200 characteristics. 37

2.4 Description of applications. 37

3.1 Comparing compiler input parameters for cache prefetching vs. I/O prefetching. 40

3.2 Application characteristics on HURRICANE. 62

3.3 Application characteristics on IRIX. 81

4.1 Description of applications. 101

4.2 Pages freed by system or by release, and pages rescued from the free list. 107

5.1 Graduated instructions for static and dynamic prefetch distance calculations (millions of instructions) 118

5.2 Properties used in presenting continuous pipelining algorithm 135

5.3 Additional procedures used in presenting continuous pipelining algorithm 135

5.4 Effect of pipelining algorithms on code size and execution time 142

5.5 Simulated performance characteristics 146

Chapter 1

Introduction

640K should be enough for anyone. — Bill Gates

Out-of-core applications are here to stay. Programs with data sets that exceed the size of main memory are easy to find, despite the exponential growth in memory capacity. Examples can be drawn from many branches of computer science including artificial intelligence, computational linguistics, databases, graphics, and scientific computing. For the "Grand Challenge" applications of science and engineering the need for more memory appears to be essentially unbounded. For instance, input data sets for scientific visualization can currently exceed 100 gigabytes [19], while the storage requirements for computational biology tasks like protein structure prediction can be measured in petabytes [57]. These types of problems have long been the driving force behind advances in supercomputing, yet as faster processors and massively parallel machines make it possible to solve problems that were computationally infeasible in the past, the demands on the memory system also scale at a commensurate pace.

For these memory-intensive applications, we can expect that disk drives will continue to form the backbone of the memory hierarchy, providing stable, cost-effective, mass storage. I/O will be required throughout the computation, both to bring the required data into main memory and to write intermediate results out to disk. Unfortunately, although advances in magnetic storage technology have led to dramatic increases in disk drive capacity and throughput, the average access time has not improved significantly. Given the large (and

increasing) gap between processor speeds and disk latencies, it is expected that out-of-core programs will spend a substantial portion of their execution time waiting for I/O. If we hope to leverage more powerful processors to solve the scientific computing problems of the future, we clearly need techniques to manage the I/O requirements of these programs.

One effective technique to reduce the time spent waiting for I/O is *prefetching*. By requesting data before it is needed, the I/O latency can be overlapped with other computation, resulting in dramatic reductions in the overall execution time. Successful prefetching, however, depends critically on determining what data should be prefetched and when the prefetch should be initiated. Because making these decisions manually represents a substantial burden on the application programmer, this thesis explores automated techniques, which are based in static compiler analysis, for obtaining the required information.

In addition to suffering from large I/O latencies, out-of-core tasks are also extremely bad neighbors in a multiprogrammed environment, yet executing them in such a setting is desirable for a number of reasons. It is not only more cost-effective to be able to share machine resources between an out-of-core application and other programs, it may also be necessary for some applications such as a visualization task that is enabling the user to interactively guide a physical simulation in real-time. Unfortunately, operating on massive data sets consumes physical resources (memory and disk bandwidth) at a rapid rate, thereby potentially displacing the working sets of other applications and increasing their page fault service times. To make matters worse, the performance gains achieved if prefetching for out-of-core applications is successful will cause physical resources to be consumed at an even faster rate, increasing the negative impact on other applications. A key observation is that the excessive resource consumption by out-of-core tasks is not the result of their inherent resource requirements. It can instead be attributed to resource management policies in the operating system that are not well suited to this class of applications. For instance, global page replacement policies perform well in the general case, but allocate too many pages to an out-of-core program that is sweeping though its data set rapidly.

Good overall performance in a multiprogrammed workload with out-of-core applications depends on making the right decision about what data should be replaced from memory to make room for the prefetched

data. Thus, in this thesis we also evaluate the use of compiler analysis to guide replacement decisions. For both prefetching and replacement, we show that static analysis alone is insufficient in many cases, and must be coupled with a run-time system capable of adapting to dynamic conditions if performance improvements are to be realized.

We begin the rest of this chapter by taking a closer look at the performance problems experienced and caused by out-of-core programs before summarizing other approaches for managing I/O. We then describe our research goals, list the major contributions of this thesis, and end with an overview of the remaining chapters.

1.1 Problems with Out-of-Core Applications

Given that applications with extremely large data sets will require I/O throughout their execution, there are two primary approaches that could be used—either manage the I/O explicitly through `read` and `write` system calls, or rely on the operating system (via paged virtual memory) to fetch and replace data as needed. These traditional solutions force the programmer to choose between performance and ease-of-use.

1.1.1 Virtual Memory Versus Explicit I/O Interfaces

Using virtual memory to handle intermediate I/O requirements has one enormous advantage over explicit I/O calls—it is easy to use. Programs that were written assuming the data set would fit in memory need no modification (other than increasing the size of relevant parameters and data structures) to scale them up to larger (out-of-core) problem sizes. Programmers writing new applications can focus on the problem they are trying to solve, without regard to the I/O requirements of their programs. For instance, algorithms can be chosen for their numerical stability, rather than to minimize the number of disk accesses required. Unfortunately, the abstraction of paged virtual memory is not transparent with regard to performance: it is painfully obvious when the physical memory boundary has been crossed. Worse, although paging clearly degrades performance, the programmer cannot easily address these problems as paging decisions are controlled by the operating system.

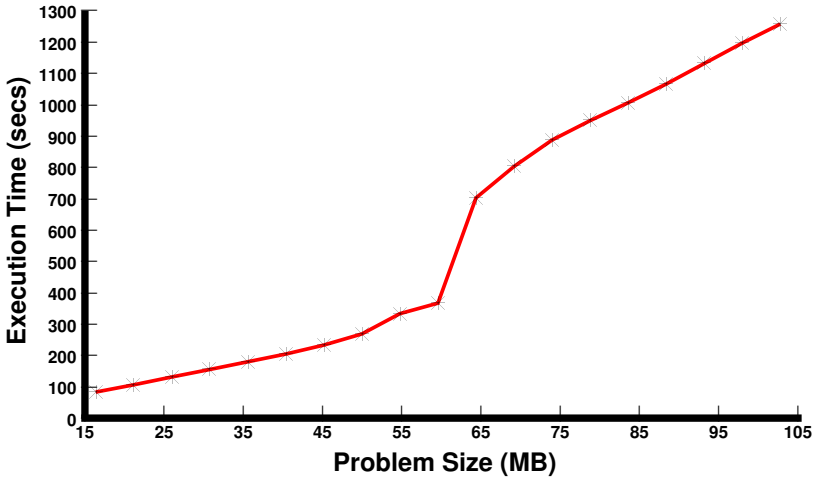
There are essentially two reasons why paged virtual memory typically has poor performance: the timing

of requests, and the size of requests. First, disk requests are initiated in response to a page fault. That is, the I/O is initiated when the application needs the data, forcing the application to stall for the full latency of a disk read before it can continue. Second, each page fault typically brings only a page-sized chunk of data into memory at a time. Even if we construct an extremely high-bandwidth disk array, and stripe our data across all the disks, we cannot exploit this extra bandwidth since only a single disk is ever active at any time.

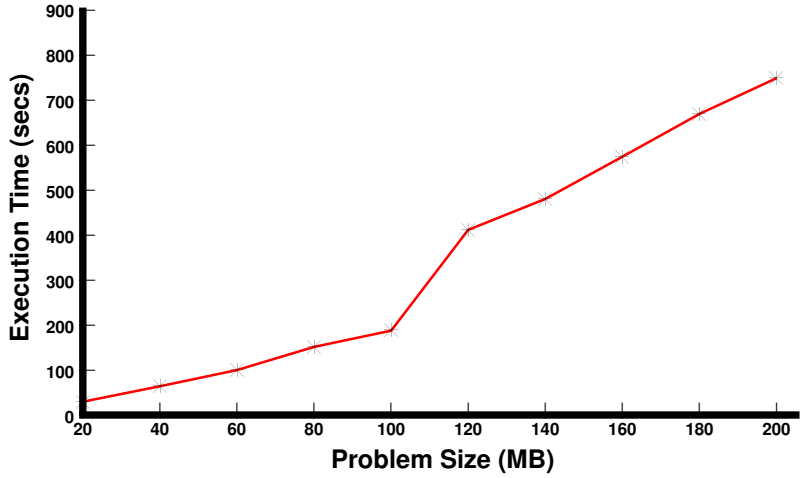
These performance problems can be addressed by explicit I/O calls as follows. First, the non-blocking I/O calls provided by asynchronous I/O interfaces allow an application to hide latency by overlapping disk I/O with computation. Second explicit I/O calls can fetch a large number of blocks in a single request, which is important to fully exploit the underlying parallelism in high-bandwidth I/O systems (e.g., disk arrays).

While explicit I/O offers the potential for improved performance over paging, it unfortunately suffers from several disadvantages. The primary disadvantage is the large burden placed on the programmer of rewriting an application to insert the I/O calls—one goal of this thesis is to avoid placing additional burdens on the programmer 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



(a) Performance of BUK on a research operating system (HURRICANE) using virtual memory for I/O



(b) Performance of BUK on a commercial operating system (IRIX) using virtual memory for I/O

Figure 1.1. Virtual memory performance on two systems for in-core and out-of-core problem sizes.

relied on paged virtual memory from the start.

1.1.2 Performance Using Virtual Memory

The simplicity of the virtual memory programming model makes it an extremely appealing interface to target, if the performance difficulties can be overcome. To illustrate some of the challenges we present an example of an out-of-core application that uses virtual memory for I/O. More details on this benchmark and the experimental setup will be given later in Section 2.4.4, with only the most relevant characteristics presented here.

Figure 1.1 shows the total execution time for a single benchmark from the NAS Parallel suite [6], across a range of data set sizes, on two very different systems. The benchmark is BUK: a bucket-sort algorithm with computation that scales linearly with the problem size. If physical memory were unlimited, we would expect the performance curve to be a straight line. Instead, on both systems a large increase in the execution time can be seen when the physical memory is exhausted and paging is needed. Results on HURRICANE, a research operating system designed for large-scale multiprocessors [56], are shown in Figure 1.1(a). On this system, every page fault pays the full cost of a disk access and the effect on performance at out-of-core problem sizes is severe.

Although it would be easy to dismiss the HURRICANE results as an artifact of a research prototype, this is not the case. Figure 1.1(b) shows the same benchmark (with different problem sizes) on a modern, commercial operating system (Irix 6.5). Despite large disk buffers, and disk controller read-ahead policies for sequential data [45], it is still easy to discern the memory boundary from the performance curve even though over two thirds of the references in BUK are sequential array accesses! The remaining accesses in BUK are essentially random, and the large gap between disk latency and processor performance makes them extremely costly. Furthermore, the disk read-ahead policies may be defeated by a lack of control over the mapping of pages to disks and the order in which page faults occur. That is, even though a particular data structure may be accessed sequentially in the program code, there is no guarantee that the accesses seen by a particular disk will also be sequential.

To successfully prefetch page faults it is necessary to determine which pages will be needed in the future. Although automatically detecting access patterns at run-time is conceptually appealing, in practice it is extremely difficult. Indirect references (such as those found in sparse-matrix codes) lead to random accesses which are not predictable by pattern detection. Even when patterns exist, as in regular strided or stencil matrix codes, interleaving of accesses from multiple data structures can obscure these patterns at run-time. Finally, as the BUK example illustrates, detecting sequential accesses is not sufficient. To make matters worse, prefetching based on pattern detection can actually hurt performance when the future accesses do not conform to the previously detected pattern. Mis-predicted prefetches are costly because they displace

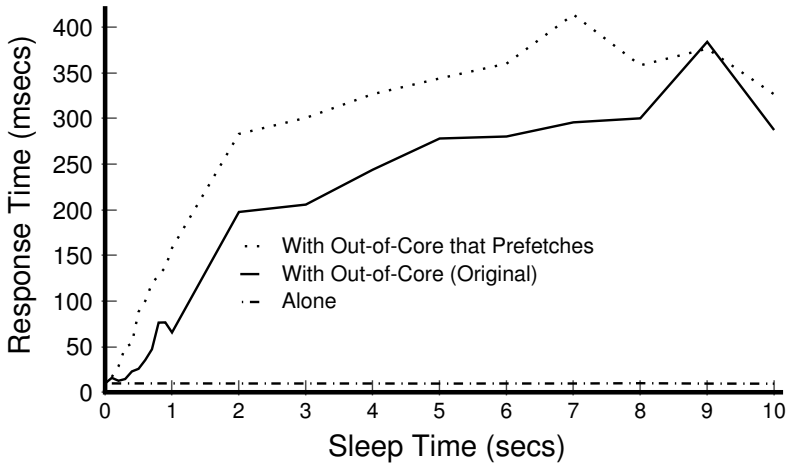


Figure 1.2. Time to touch 1MB of data for varying *think times* between accesses.

other pages from memory without providing any benefit. Smith refers to this effect as *memory pollution* and observes that it worsens with increasing page sizes [48].

This thesis presents an alternative method for predicting which pages will be needed in the future based on static analysis techniques that have the potential to automatically detect access patterns at compile-time. Given knowledge of the access pattern, it is also possible to statically transform the program to insert memory management primitives, such as prefetch instructions for data references that are expected to incur page faults. Such compiler-based techniques have been successfully applied in the past to the problem of predicting cache misses and prefetching data from main memory into the processor cache [39, 41]. Since physical memory can be viewed as just another level in the memory hierarchy, it is natural to consider applying these methods to the problem of prefetching data from disks into memory. Section 2.3 presents our design for a system that combines static analysis with dynamic adaptation, while Chapter 3 evaluates how well this design improves the performance of out-of-core applications on a dedicated machine.

1.1.3 Impact on Other Applications

As noted earlier in this chapter, out-of-core applications can hurt the performance of other programs by displacing their working sets from memory. The negative effect of out-of-core applications on other applications is most distressing for interactive programs where a human user is able to perceive an increase in

response time. Ironically, these interactive programs are also particularly susceptible since they spend most of their execution time waiting for user input, and are unable to defend their memory pages. Figure 1.2 shows the results of a simple experiment that demonstrates the magnitude of the problem.

In this experiment, we simulate the memory reference behavior of an interactive task by executing a simple program that responds to *requests* by touching a portion of its data set. Requests are separated by some fixed amount of *think time* representing delays between input from a user. The measured response time is the amount of time required to touch the data set, and should be a constant independent of *think time* if the pages of the data set remain in memory between accesses. This simple program is executed both (i) alone on the machine and (ii) with an unmodified out-of-core program (in this case, a large matrix-vector multiply that consumes memory rapidly).

As can be seen from Figure 1.2, when the interactive program executes on a dedicated machine, its response time is indeed uniform as *think time* ranges from 0 to 10 seconds (see the line labeled "*Alone*"). When a non-prefetching out-of-core program is executed concurrently, however, the response time increases dramatically, even for relatively short *think times* of 1-5 seconds, as shown by the "*With out-of-core*" line. Employing prefetching to improve the performance of the out-of-core program only serves to make the situation worse for the interactive task.

In this thesis we argue that the responsibility for reducing the negative impact of the out-of-core program should rest with the out-of-core program itself, since it is the source of the problem. By leveraging the same static analysis that supports prefetching decisions, the compiler can identify pages that may be good candidates for replacement, thus reducing the need to steal pages from other applications. The effect of explicitly releasing memory on the out-of-core program is studied in Chapter 3, while the overall effects in a multiprogrammed environment are the subject of Chapter 4.

1.2 Related Work

The general problems of hiding disk latency through prefetching, and improving the memory performance of data-intensive applications have been well studied in the past in a variety of forms. In this thesis we

focus on techniques with a strong operating system or compiler component; although there is a large body of research on algorithms for out-of-core computations, we will not discuss them here since we are not modifying the program's algorithm. Our techniques can be applied to hide the I/O latency in an out-of-core program after the most suitable algorithm for the problem has been chosen.

Related research can be classified into five broad areas: (i) prefetching of page faults in virtual memory systems; (ii) prefetching of explicit file accesses; (iii) combining prefetching and replacement decisions; (iv) compilation for out-of-core programs; and (v) application-controlled memory management. We will discuss each of these areas in turn.

1.2.1 Prefetching in Virtual Memory Systems

Over twenty years ago, Trivedi presented a strategy for automatically extracting application access patterns in looping programs using a compiler [55]. The detected patterns were used to implement *prepaging*, which included directives to fetch particular pages into physical memory, and replace other pages. At a high level, this work is very similar to the scheme presented in this thesis, with two important differences. First, advances in compiler technology have greatly expanded the range of applications that can be handled. 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 [41]. Second, the introduction of a run-time layer to filter unnecessary hints allows the compiler to act much more aggressively, further increasing the situations in which prefetching and releasing memory can be applied successfully.

Other work in the area of prefetching for paged virtual memory systems generally depends on the operating system being able to detect patterns to initiate prefetching. Curewitz, Krishnan, and Vitter investigated using techniques developed for data compression to predict what to prefetch [20]. In their design a *prefetcher* examines a database client's previous page requests and issues prefetch requests to the database server. They note that prediction accuracy is better when a separate prefetcher is used for each instance of a client application, since user access patterns can vary widely. In a similar vein, Song and Cho use the fault history to detect

patterns and initiate prefetching [49]. Pattern-based prediction techniques all suffer from the fact that some number of faults are required to establish patterns before prefetching can begin. Worse, mis-predictions are likely to occur when the patterns change, leading to potentially harmful memory pollution.

Much of the work on automated page fault prefetching has been done in the context of distributed shared memory systems, where pages are fetched from remote memories rather than from local disks. Representative examples include [7, 50, 63]. Network latencies in these settings are typically much smaller, and hence easier to hide, than disk latencies. The fundamental issue is that mis-predictions increase with the latency. Techniques which are successful at hiding network latency are likely to cause too much memory pollution when applied to the problem of hiding disk latency. In contrast, techniques that are successful for disk prefetching can also be used to hide smaller network latencies in a distributed shared memory system.

A final option for virtual memory prefetching makes use of the *madvise* system call, provided by some UNIX operating systems. This system call allows a programmer to give advice such as MADV_WILLNEED or MADV_DONTNEED for ranges of the virtual address space. While this is conceptually similar to the operating system interface we present, it relies on the programmer to manually provide information about the memory usage of their program. In addition, even when the interface is provided, many operating systems make no claims about how the advice will be handled—it may simply be ignored. Thus the utility of the interface, by itself, is quite limited.

1.2.2 Prefetching for Explicit File Accesses

Prefetching in file systems by automatically detecting file access patterns has been well studied [3, 24, 25, 28, 32, 34]. 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 [34]. Griffioen and Appleton construct a *probability graph* based on prior file system accesses [24]. 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, and/or have many accesses to the swap file. Kotz and Ellis studied prefetching

when a program's dynamic access patterns matched one of eight access patterns commonly found in parallel applications [32]. Once again, techniques that depend on being able to detect an access pattern cannot prefetch until the pattern has been established and may do the wrong thing when the pattern changes. In addition, the access patterns may be extremely difficult to detect automatically, or may not match the pre-selected set of possible patterns.

Another technique implemented in file systems is to support prefetching based on information supplied explicitly by the application. One such approach is the TIP system, developed by Patterson et al. [43], which considers both prefetching and caching, and will be discussed in the following section.

1.2.3 Integrating Prefetching and Replacement

The importance of considering both prefetching and replacement decisions in tandem has been studied by several groups in the context of I/O prefetching and caching for file system references. Cao *et al.* [11] present several properties that optimal prefetching and caching strategies must have, however the complete reference stream is required to satisfy these properties. The TIP system for I/O prefetching by Patterson *et al.* [43] uses a cost-benefit model to estimate which file blocks should be replaced from the buffer cache, based on access-pattern hints disclosed by the application. While the goal of using application-specific knowledge to improve overall system performance is the same as in our system, we focus on virtual memory references rather than file reads and writes. Because it is much more costly to track all virtual memory references (versus explicit file requests only) the techniques applied by the TIP system for determining what to eject from the file cache are not especially applicable for virtual memory management decisions.

In the original TIP implementation, applications had to be manually modified to generate the necessary access hints. Recently, another approach for automatically modifying applications (using a binary modification tool on the program executable) to provide hints about their future accesses has been presented by Chang and Gibson [12]. Applications are altered to speculatively execute their code for the purpose of discovering future read requests, which can be passed as hints to the TIP system.

Vellanki and Chervenak used cost-benefit analysis similar to TIP, however prefetch decisions were made

using a markov-based predictor built from past disk access history [58]. Their scheme suffers from the same limitations as other approaches that use past access patterns to derive future predictions. Finally, Voelker *et al.* present a hybrid approach that uses both disk and network memory prefetching [59]. They assume that future accesses are fully known (either through TIP-style application hints, or some other method) and adapt the *Forestall* algorithm of Kimbrel *et al.* to consider both types of prefetching. Although their results demonstrate that prefetching can be successfully applied to hide multiple kinds of latency at the same time, they do not address the question of deciding what to prefetch when full knowledge of future accesses is not available.

1.2.4 Compiling Out-of-Core Programs

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 [9]. The second area is inserting explicit I/O calls into array codes [16, 31, 42, 54]. 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 [9], while we aim to improve the performance of 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. However, asynchronous I/O calls cannot be issued unless it is known that the data being read will not be modified between the asynchronous call, and the point of the original read. Aliasing can complicate the compiler's ability to determine the safety of changing synchronous reads into asynchronous ones at an earlier time, and limits the opportunities available for applying this transformation. In contrast, prefetching based on a virtual memory interface can be applied at any time, even if all aliases cannot be resolved.

1.2.5 Application-controlled Memory Management

Many researchers have suggested that better performance can be obtained if sophisticated applications are given control over their own memory management decisions. Most previous work in this area has focused on how the operating system can provide this functionality to the applications. For instance, the Mach

operating system supports external pagers to allow applications to control the backing storage of their memory objects [44]. Extensions to the external pager interface have been used to implement user-level page replacement policies [38] and to support discardable pages (i.e., dirty pages that do not have to be written to backing store) [51]. More aggressive application control of physical memory was implemented in the V++ kernel by Harty and Cheriton [27]. In their scheme, the application was given complete control over a cache of physical pages, enabling the implementation of application-specific memory management policies. Giving applications more control over physical resources (not just memory) is also a part of the motivation behind extensible operating systems such as Exokernel [30], SPIN [8], and Vino [47].

Providing support for application-specific control is only half of the picture, however. If the mechanisms provided require programmers to re-write their applications manually, the full power of the scheme is unlikely to be realized in the real world. In contrast, our approach provides not only the mechanisms for application-controlled memory management, but also a means to leverage these mechanisms automatically through the use of the compiler.

1.3 Research Goals

At the highest level, the goal of this thesis is to demonstrate the following statement:

> **A combination of I/O prefetching and page replacement hints inserted by automatic compiler analysis and transformation can successfully hide the latency of page faults in out-of-core programs while simultaneously reducing the negative impact on the performance of other programs in a multiprogrammed environment. Adaptation to run-time conditions can be achieved by a user-level software layer which synthesizes information provided by the compiler and the operating system to perform memory management operations (i.e. issuing prefetch and replacement hints) only when they are actually needed.**

We aim to demonstrate the preceding statement by providing a concrete implementation of the technique presented by this thesis, using a full-featured optimizing compiler and a modern commercial operating system.

1.4 Contributions

This thesis makes the following contributions:

- The proposal of a new strategy for coping with I/O latency in out-of-core programs that introduces a run-time layer to integrate information obtained from compiler analysis with information readily available to the operating system during program execution. Key features of this strategy are that it is fully-automatic, capable of improving overall performance in a multi-programmed setting, and able to adapt to dynamic conditions.

- A detailed evaluation of the proposed strategy, based on a full implementation of each of the three key components. The prefetching algorithm is implemented in the SUIF optimizing compiler [26, 61]. The operating system support has been implemented in two distinct operating systems—a research prototype designed for large-scale multiprocessors (HURRICANE) and a commercial operating system (Irix 6.5). By comparing the results on the two systems, we are able to gain insight into the key features that the operating system support should provide, and the effect that other architectural and software design decisions may have on our ability to support out-of-core programs successfully.

- An investigation into the feasibility of having out-of-core applications voluntarily release pages to improve overall performance in a multiprogrammed environment. Typically replacement decisions are the sole responsibility of the operating system, which has the only global view of competing demands for memory. This thesis evaluates whether it is possible for applications to make local decisions that improve global performance, based on minimal information about memory usage from the operating system.

- The development, implementation, and evaluation of a new compiler algorithm for *software pipelining* which is able to handle unknown or variable latencies, multi-dimensional loops, and unknown loop bounds (all of which present problems for existing software pipelining algorithms).

Although this thesis presents and evaluates a solution to a particular resource management problem (that of managing physical memory for an out-of-core program), we believe that the approach taken here, bringing together the strengths of the compiler and the operating system with a run-time adaptation layer, will be applicable to many other resource management problems as well.

1.5 Overview of Dissertation

Chapter 2 lays out the overall design of the system and describes how the compiler, operating system, and run-time layer relate to each other. It also describes the benchmarks and hardware platforms used to evaluate the system. Chapter 3 studies the effect of our approach on the performance of out-of-core programs running on a dedicated computer system. Both the initial research prototype operating system, and the commercial system are studied. Chapter 4 studies the performance of both out-of-core and interactive applications in a multiprogrammed setting. Based on these results, a number of limitations of the compiler scheduling algorithm are identified, leading to a new algorithm which is described and evaluated in Chapter 5. Finally, Chapter 6 summarizes the important results in this dissertation, highlights their implications, and suggests directions for future work in this area.

Chapter 2

System Design Alternatives and Evaluation Framework

Recognizing the need is the primary condition for design. — Charles Eames

In this chapter we present the basic design of our system for automatically managing the I/O requirements of out-of-core programs. In this thesis, we focus on scientific computing applications, where large arrays are the primary data structure, and most data accesses occur in loops. Our goal is to fully hide I/O latency, thus eliminating its impact on overall execution time for this important class of applications. Conceptually, one can view our approach as enhancing the performance of virtual memory, since that is the abstraction we present to the programmer. Our design space, however, is not limited to the operating system where virtual memory support is traditionally implemented. We view the problem as one of collecting enough information about an application's memory requirements to enable us to make good management decisions. Relevant information may be most readily available at different times in a program's life cycle, and to different entities. Similarly, some actions or transformations may be easier to perform at certain program stages than at others.

In this chapter, we explore alternatives for collecting and acting on the information needed to make good memory management decisions. We begin in Section 2.1 by considering the features and actions our system

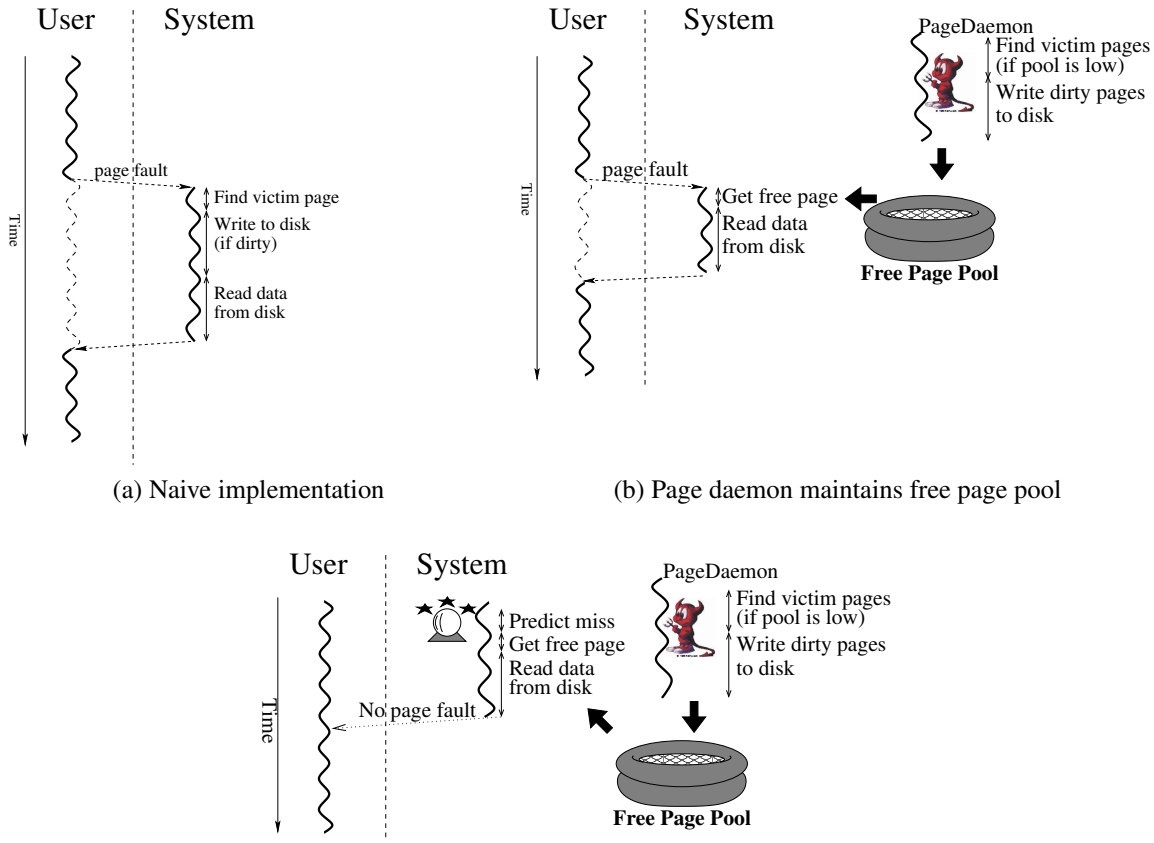
needs to support. We then present a number of options for providing these features, identifying strengths and limitations of each in Section 2.2. The actual design of the system presented in this thesis is then outlined in Section 2.3. The chapter concludes in Section 2.4 with an overview of the framework we will use to evaluate our overall design.

2.1 Key Requirements

In Section 1.1.1 we identified two performance problems with virtual memory, namely the timing and size of I/O requests. A third problem is that the operating system must evict data from memory without knowing when or if the application will need it again. Any design that aims to preserve the abstraction of virtual memory for the application programmer must address these three problems.

Under paged virtual memory, the operating system performs two types of disk accesses on behalf of applications: (i) pages are read from disk to satisfy a reference to data that was not present in memory, and (ii) modified (or *dirty*) pages are written out to disk to create free page frames for these reads. In the earliest implementations of virtual memory systems, victim page frames were selected and written out to disk (if dirty) in response to a page fault, creating a free page frame that was immediately filled by reading the required data from disk [22]. With this model, the performance cost of a page fault could be the latency of two disk accesses, as well as the processing cost of selecting a victim frame. Figure 2.1(a) illustrates the situation.

In general, hiding write latency is straightforward since writes can be buffered and pipelined, however when cleaning dirty pages, the write must complete before we can reuse the page frame. Fortunately, since the writes are transparent to the application, there is no restriction on when the request can begin. Modern Unix systems hide write latency using the *page daemon* approach introduced by 3BSD UNIX [4]. Rather than waiting for a free page to be needed before finding one, the operating system maintains a pool of (clean) free pages which can immediately be allocated in response to a page fault. A background system process (called the page daemon) monitors memory usage, identifies victim pages, and schedules writes for those that are dirty before they are moved to the free list. Figure 2.1(b) shows how the use of a page daemon improves



(a) Naive implementation

(b) Page daemon maintains free page pool

(c) Pages are prefetched, eliminating page faults

Figure 2.1. Improving demand paging performance

performance.

The page daemon illustrates a key principle that we need to exploit to further improve virtual memory performance: the operation (a write) is scheduled early enough that the result (a clean free page) is available before it is needed, and furthermore the operation occurs in parallel with normal execution of the application. In other words, the introduction of the page daemon solves the timing problem for virtual memory writes by decoupling the I/O activity from the application's need for page frames.

The page daemon is also able to address the request size problem for virtual memory writes. Because free page frames are no longer generated in response to a particular page fault, the daemon can collect a larger set of dirty pages and write them to disk as a single large request, thus making more effective use of the available disk bandwidth. There is, of course, a cost associated with producing free page frames before they

are needed—the number of "soft" page faults could increase, and frequently-modified pages could be written to disk repeatedly they are cleaned and freed too early.

In contrast to writes, hiding *read* latency is difficult because the application cannot proceed without the missing data and thus must stall waiting for the I/O to complete. As with writes, we need to find a way to schedule the read operation early enough that the data arrives in memory before it is needed. Thus, 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. This is an especially difficult task for a virtual memory system since the actual read request occurs as a side effect of attempting to use a particular data item. Our system needs to provide a mechanism for predicting when the application is likely to access a page that is missing from memory so that an asynchronous *prefetch* request can bring the required data into memory before it is needed. Our prediction mechanism should attempt to bring pages into memory "just-in-time" to minimize the added memory pressure as much as possible. Figure 2.1(c) illustrates the effect on performance of an ideal predictor which brings pages into memory before they are needed, eliminating page faults in the application. We will refer to the combination of predicting pages that should be brought into memory, and deciding when to initiate the I/O requests for these pages as *prefetching*.

Prefetching does not reduce the number of disk accesses; it simply attempts to perform them over a shorter period of time by overlapping I/O with normal program execution. Thus, it cannot significantly 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 [13, 33, 53]. Roughly speaking, one can always increase the I/O bandwidth by purchasing additional disks. It is important, therefore, that our prefetching mechanism be able to support multiple simultaneous requests to allow us to exploit the available bandwidth in a multiple-disk system. By overlapping I/O requests with each other, as well as with program execution, we create the potential for speedup factors greater than two.

In addition to hiding read latency and leveraging the available 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 (least-recently-used) 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 need a mechanism to predict which pages are relatively less important to a particular application, and to cause these pages to be freed first. Conceptually, we would like the predictor of Figure 2.1(c) to be responsible for replenishing the free page pool, based on its oracular knowledge of program accesses. In essence, we would like the flexibility to define an application-specific replacement policy, rather than being forced to choose from a fixed set of pre-defined policies.

In summary, the design of a system to improve virtual memory performance must provide the following features:

- support for *prefetching*, that is, asynchronously requesting data before it is needed, thus allowing us to tolerate disk read latency

- support for *multiple outstanding requests* to exploit the strengths of high-bandwidth multiple-disk I/O systems

- a means to explicitly identify pages that can be replaced, allowing us to build *application-specific replacement policies*.

Entity	*Time*	*Information*	*Limitations*
Programmer	application development	purpose of program	not system expert
Compiler	compile-time	data structures, control flow, target system	no dynamic information
Operating System	run-time	global resource usage, dynamic behavior	no knowledge of program

Table 2.1. Available Information Sources

2.2 Design Alternatives

Having identified the features our system needs, we now discuss various alternatives for providing them. Our most difficult challenge is to predict page references accurately, allowing missing pages to be brought into memory before they are needed, and non-essential pages to be replaced first. In Section 2.2.1, we consider three possible sources of information to drive our predictions: the application programmer, the compiler and the operating system. We also consider the opportunities for making prefetching and replacement decisions in each of these components. To make the best decisions, we would like to combine the information that is readily available to each component in a single place; options for sharing information are presented in Section 2.2.2. We next consider how our predictor might communicate its decisions to the operating system, which has ultimate responsibility for managing physical memory pages, in Section 2.2.3.

2.2.1 Potential Sources of Information

Table 2.1 summarizes the entities that could be involved in collecting information about a program's memory accesses and implementing prefetching and replacement decisions.

The Programmer

The application programmer has sole responsibility for high-level decisions such as the choice of algorithm to solve the problem, the data structures, and the overall program structure. Given suitable interfaces to the operating system, the programmer could make memory management decisions manually as the program is being designed and written. Explicit I/O calls are a typical example of this approach, which has three major drawbacks. First, the programmer is likely to be an expert in the application area, not the details of the target system where the program will run. Second, the decisions of the programmer are built into the source code

and may seriously restrict the portability of the program; operations that improve performance on one system may be unsupported or may degrade performance on another. Third, programs that were initially written without concern for I/O cannot be improved without substantial programmer effort.

We believe that producing correct, readable, and maintainable programs is already difficult enough, and that limited human attention is best spent on these high-level details, rather than on contorting the program's source code to optimize performance on a specific execution platform. Our goal in this thesis is to allow the programmer to use the abstraction of virtual memory, without concern for performance. Therefore, we do not consider the programmer as a viable source of information about memory access patterns.

The Compiler

The compiler is responsible for translating the application from its high-level representation into a binary executable for a target system. It thus has natural access to information about data structures and control flow, such as the arrays and loops that are common in scientific computing applications. The compiler also has knowledge of low-level details of the target architecture. Although these architectural details are typically only used by the back-end to select machine instructions and allocate registers, it is reasonable to consider augmenting the compiler with system-level information such as the page size. Given this information, it is straightforward for the compiler to detect array accesses in loops, to determine which of these accesses will fall on separate pages, and to detect when pages are being reused. To predict whether or not these accesses will cause page faults, however, the compiler also needs to know how much data is being accessed in the loop, and how much memory is available on the target system. Accurately determining these parameters at compile-time is not always possible, even for numerical applications, since loop bounds may be symbolic variables at compile-time and available memory depends on dynamic demand from other programs at run-time. General-purpose applications are even harder to analyze statically, and are beyond the scope of this thesis.

Assuming predictions about page faults can be made at compile-time, the compiler could insert new instructions to prefetch these pages. To hide the latency of a disk access on modern processors, however,

the prefetch instruction needs to occur millions of instructions before the actual data reference. For general-purpose codes with arbitrary control flow, it is impossible to find a single, static point in the code to insert the prefetch so far ahead of the use of the data. The loop structure common in numerical codes, on the other hand, gives us some chance for success. Even for numerical applications, however, deciding where in the program to place the prefetch instructions requires the compiler to know how long it will take for the prefetch to complete on the target system (and how long it will take to execute the code between the point where the prefetch is inserted and the point where the access occurs). While it is possible to provide the compiler with an estimate of the time to read a page of data from disk into memory, the actual time will depend on dynamic conditions such as other requests at the disk.

In spite of being limited by a lack of dynamic information, the compiler has considerable advantages. First, it does have access to some high-level program information (less than the programmer, but certainly much more than the operating system). Second, the time spent analyzing the program and collecting information about accesses only occurs once, while the payoffs in improved program performance will be realized each time the program is run. Third, the transformations performed by the compiler are fully-automatic, making it easier to port the application to a different system. We want to leverage the strengths of the compiler by making decisions *statically* whenever possible, but we need to retain the ability to improve upon these decisions *dynamically* as better information becomes available during execution.

The Operating System

The operating system is responsible for allocating system resources to competing applications, and is the only entity with a global view of dynamic conditions. These characteristics make it the natural choice for deciding when to allocate more memory to an application for prefetching, and when to reclaim memory from an application. Unfortunately, the operating system has no higher-level information about the data structures or control flow of a program. It must base its decisions on observations of past behavior, with the typical assumption that the recent past is a good predictor of the near future.

To make better decisions, we would like to combine the dynamic information available to the operating

system with the application-specific information available to the compiler.

2.2.2 Options for Sharing Information

Rather than trying to make any memory management decisions with incomplete information, we could have the compiler insert code to periodically notify the operating system of future page accesses by the application. For instance, a call could be added at the start of a set of loops to pass down a list of virtual pages that would be accessed in the loop. The operating system would then decide which of these pages needed to be prefetched and schedule the disk reads appropriately. Similarly, the list could be used to select pages for replacement (either pages that are not in the list, or pages whose access occurs furthest in the future)[1]. The problem is that the operating system does not know how much computation time will occur between accesses to pages in the list, and thus it doesn't know when to schedule the disk reads. The compiler could include a time estimate for each page, but for complex access patterns, the amount of information being communicated and the work required by the operating system to act on this information easily becomes excessive. For example, in the bucket sort application (BUK), many of the important data accesses are indirect references based on the contents of a very large array. The values in this array are unknown at compile-time, hence the list of page addresses would need to be generated dynamically and the information passed to the operating system would be at least twice the size of the array itself.

An alternative is to have the compiler make the best decisions it can based on available information. Those decisions (i.e., requests to prefetch or free pages) are re-examined at run-time when better information becomes available. For instance, a request to prefetch a page can be discarded if the page is already in memory, or if there is not enough memory available for the request. A request to free a page may be delayed or discarded if memory is ample when the request occurs. Although this information belongs to the operating system, crossing the user-level/system boundary is an expensive operation. The overhead of consulting the operating system for a decision on each request could become significant if the compiler frequently inserts requests which are discarded at run-time. We could force the compiler to only insert requests that are highly

[1]This is essentially the approach taken by the TIP [43] system for prefetching explicit `read` requests. The original `read` system calls are used to track the application's progress and schedule the prefetches; with virtual memory accesses we do not have an original system call to use for this purpose.

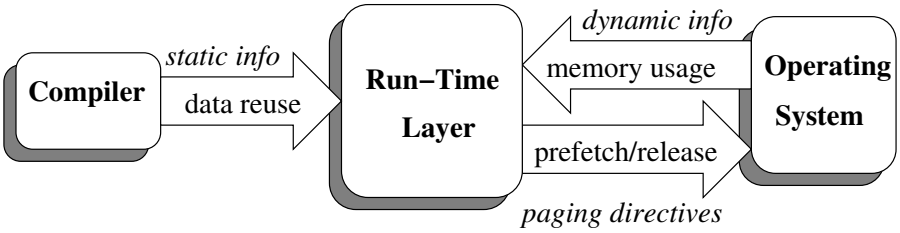


Figure 2.2. Information flow between components of our system.

likely to be needed, but given the limited information available at compile-time, this approach would lead to many missed opportunities for prefetching, limiting our ability to hide I/O latency. Instead, we introduce a new player to our system: a *user-level run-time library* (we refer to this library as the run-time layer). Figure 2.2 illustrates the basic flow of information between the components in our system.

The Run-Time Layer

The purpose of the run-time layer is to intercept the requests inserted by the compiler and decide whether or not they should be passed to the operating system. To make these decisions, it needs some information from the operating system itself. Specifically, the run-time layer needs to know which virtual pages of the application's address space are in physical memory and how much memory is currently available. Using traditional approaches, the run-time layer could obtain this information by making a system call when it needs updated data, or having the operating system make an upcall into the application whenever the information of interest changes. Since the information on memory usage is expected to change frequently, and the primary purpose of the run-time layer is to avoid frequent user/system boundary crossings, neither of these options are suitable. Our solution is to create a shared data structure that is maintained by the operating system and read by the run-time layer. This data structure is similar in spirit to the `/dev/kmem` special file, however it is read-only, available to any user and specialized for our purposes. The basic idea is to create a map of the virtual address space so that the run-time layer can determine whether a particular page is present in physical memory or not. This binary decision requires only one bit of information per virtual page, thus, we can logically represent the shared virtual address space map as a bitvector in which bits are turned on (or set to "1") when pages are in memory and turned off (or set to "0") when pages are not in memory. The

shared data structure is also used to hold information about the number of free pages and the size of the application's resident set. We discuss the implementation of the shared data structure, including details of the virtual address space map, for both of our experimental systems in Sections 3.2 and 3.3.

A secondary benefit of the run-time layer is that it provides us with considerable flexibility in choosing and modifying the interface used by the compiler-inserted requests, while allowing us to keep the interface with the operating system as clean and simple as possible. We now consider several options for the run-time layer to communicate the results of its decisions to the operating system.

2.2.3 Options for Communicating Decisions

The run-time layer needs a way to ask the operating system to initiate reads for pages that should be prefetched, and to identify pages that could be reclaimed. We begin by studying whether any of the existing interfaces to the operating system would be suitable for this purpose. If we can target an existing interface, we can avoid both broadening the system call interface and adding new functionality to the operating system.

Asynchronous I/O Interfaces

The asynchronous I/O model is attractive because it appears to match what we need for a prefetch: a mechanism for requesting data to be brought into memory and allowing the application to proceed without waiting for the read to complete. Unfortunately, it is not a good match for the virtual memory model that we want to provide to the application programmer. Asynchronous read calls require a file from which the data is being read, and a buffer into which to read the data. Virtual memory accesses do not generally have a regular file to which a read call could be directed (memory mapped files are an exception). We could have the compiler insert code to create new files and map them to regions of virtual memory we are interested in. Prefetch requests could then be translated into asynchronous reads from the file into the appropriate user virtual addresses and release requests would become a write back to the file. This idea creates more problems than it solves, however.

There are at least three reasons why existing read/write I/O interfaces are unacceptable for our purposes. First, we need to deal with issues of aliasing and program correctness when inserting the requests at compile-

(a) Original Code

```
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];
  }
}
```

(b) Read/Write Interface

```
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));
}
```

(c) Prefetch/Release 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));
}
```

Figure 2.3. Example illustrating the importance of non-binding prefetches .

time. For the compiler to successfully insert prefetches early enough to hide the large latency of I/O, it is essential that prefetches be *non-binding* [39]. 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 the corresponding load, the value seen by the load will be stale. Hence we cannot insert a binding prefetch before 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"[2]). Since an asynchronous read call implicitly renames data by copying it into a buffer, it is a binding prefetch. Existing asynchronous I/O systems do not support other accesses to the buffer being filled until the I/O has completed (both SUN and SGI's implementations say the behavior is undefined in this event [46, 52]). To illustrate this problem, consider the code in Figure 2.3(a). If we use the asynchronous I/O interface, we might generate code similar to Figure 2.3(b). Unfortunately, this code produces an undefined

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])`).

A second problem is that there is no real integration between the operating system's virtual memory paging and the requests inserted by the compiler. For instance, the system could decide to reclaim a particular page (writing it out to the mapped file) before the explicit release request. The release request would then cause the data to be paged back into memory so that it could be written out again! Even if the release request can successfully write out the page without causing extra I/O, we have no way of telling the operating system that the page is no longer needed. This problem hints at the third major limitation of using an asynchronous I/O interface: it compels the operating system to perform the I/O request. 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.

We need a non-binding prefetch mechanism that is more tightly-coupled with the virtual memory abstraction used by the application programmer.

Slave Threads

If the operating system allows multi-threaded applications to use multiple kernel thread contexts, we could achieve a prefetch by simply passing the page address to a slave thread in the same address space and having it attempt to read from that address. If the page is not in memory, the slave thread will suffer a page fault, effectively causing the data to be prefetched. This solution is simple, non-binding, and perfectly integrated with the virtual memory abstraction. Problems remain, however. As with asynchronous read calls, the operating system has no control over whether to service prefetch requests or ignore them—it cannot tell the difference between a page fault caused by a slave thread to force a prefetch and a page fault caused because the application needs the data. We could use a special instruction in the slave thread (such as a load to register R0 on MIPS processors) and have the page fault handling code check for this case to detect a prefetch. Unfortunately, this idea would require *every* page fault to pay the price of checking for a prefetch. A second problem is that the prefetched page translations would consume scarce TLB resources well before

they are needed. Finally, we would still have no way of specifying pages to be released.

Shared Data Structure

Having introduced a shared data structure to solve the problem of efficiently passing information from the operating system to the run-time layer, we could consider using the same data structure for communicating prefetch and release requests to the operating system. This saves us from adding any new system calls, but we still have to augment the operating system to read from the data structure and process the requests. We also need to give the user-level write permission on the structure, which opens a new set of safety and security issues, as well as coordinating updates by the operating system and the run-time layer.

UNIX `madvise` System Call

Many UNIX systems provide a `madvise` system call, first introduced by 4.4BSD UNIX. This system call provides hints such as "`MADV_WILLNEED`" and "`MADV_DONTNEED`" to tell the operating system about ranges of the virtual address space that the application will or won't need in the future. Conceptually, this system call provides the interface we want: it is advisory, non-binding, integrated with the virtual memory abstraction, and can specify both prefetch and release actions. The only problem is that different UNIX variants treat this advisory call very differently (ranging from treating the advice as a command, to ignoring it entirely). To avoid confusion with existing programs, and to experiment with how the operating system should handle these calls, we prefer to introduce a new pair of system calls: `prefetch` and `release`.

2.3 Actual System Design

Having the discussed the various options available, we now present the actual system design evaluated in this thesis and discuss the roles of the three major components—the compiler, the run-time layer and the operating system—in more detail. The connections between these components are illustrated in Figure 2.4, which shows how an ordinary application is automatically transformed and managed in our system.

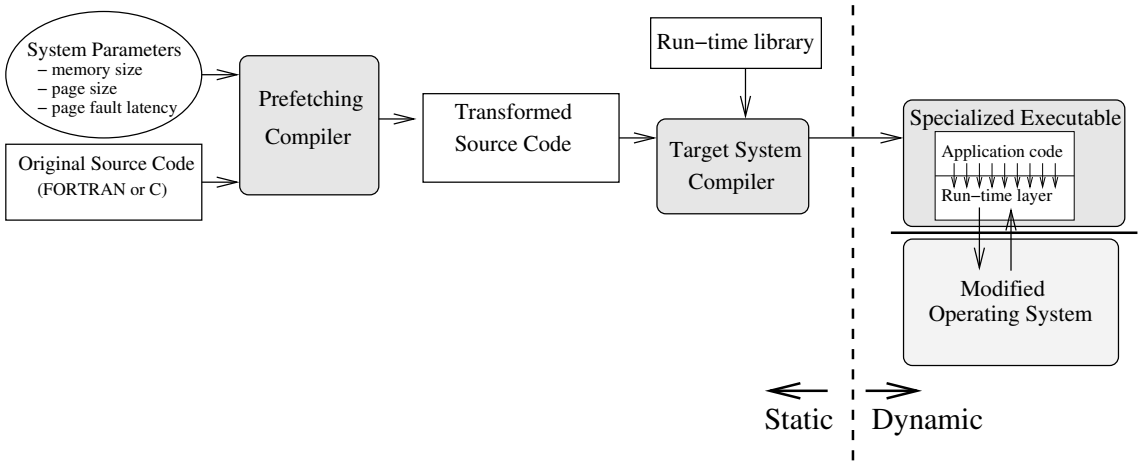


Figure 2.4. Steps in the automatic transformation of original application into prefetching/releasing executable.

2.3.1 The Compiler

The bulk of our compiler algorithm is a straightforward extension of one that was developed earlier for prefetching cache-to-memory misses in dense-matrix and sparse-matrix codes [39, 41]. 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 our implementation of each of these steps in the SUIF compiler, and the changes needed to properly support I/O prefetching are described in Section 3.1.

The static prefetching decisions made by the compiler may be inappropriate if the amount of available memory or the time to service a page fault are vastly different from the parameters used at compile-time (i.e., prefetches may not occur often enough if there is less available memory, or they may not occur early enough if the I/O latency is greater than expected). Due to the high cost of a non-prefetched page fault, we try to err on the side of inserting too many prefetches rather than too few. We cannot afford to ignore the cost of unnecessary prefetches, however.

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 [39, 41]. To address this problem, compiler algorithms have been developed for inserting prefetches only for those references that are likely to suffer misses. An analogous situation exists with I/O prefetching, since we do not want to prefetch data that already resides in main memory—hence, we perform similar analysis in our compiler (as we discuss later in Section 3.1). Unfortunately, it is considerably more difficult to avoid unnecessary prefetches with I/O prefetching for two reasons. First, we deliberately make conservative assumptions about the amount of memory that will be available at run-time. This decision leads to fewer non-prefetched pages, but has the potential to also increase the number of pages that are prefetched unnecessarily. Second, our compiler analysis considers each set of nested loops independently. This strategy is usually sufficient to capture the locality in caches. Main memory is much larger than a cache, however, and can retain data across multiple sets of loops. As a result of these factors, 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 (since unnecessary prefetches are dropped as soon as the cache tag check indicates that the data is already in the cache), the overhead of an unnecessary I/O prefetch is considerably larger. Without any additional support, our prefetch requests must make a system call and check the status of the page table before discovering that the prefetch can be dropped. The introduction of the run-time layer in our system allows us to greatly reduce this overhead by keeping 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 as we will show in Section 3.2.2.

Generating code that is fully adaptable to the range of conditions that could occur during execution is beyond the scope of this thesis, however Chapter 5 discusses techniques to increase the adaptability of the compiler support.

2.3.2 The Run-Time Layer

The run-time layer intercepts the requests inserted by the compiler and decides whether they should be passed on to the operating system. 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. Logically, each bit in the vector represents one virtual memory page. Bits are turned on (i.e., set to "`1`") by the operating system to indicate that the corresponding page is in physical memory; bits are turned off (i.e., set to "`0`") by the operating system to indicate that the page has been reclaimed. 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. Essentially, the run-time layer uses the state of memory at the time of the prefetch request (as represented by the bit vector) as a simple predictor of the future state of memory when the data reference will occur. That is, we predict that pages already in memory at the time of the prefetch will remain in memory until the data reference. In many cases this simple test can eliminate the cost of a system call, thus substantially reducing overhead.

The run-time layer also uses the bit vector to filter *release* requests for pages that are not present in memory. In Chapter 4 we examine how the run-time layer can also use information about available memory to delay and prioritize release requests inserted by the compiler.

Although it would be possible for the run-time layer to approximate the state of main memory entirely at the user level by turning bits on for prefetched pages and turning bits off for released pages, the run-time layer would be unaware of pages brought into memory by non-prefetched page faults, or reclaimed by the operating system. To give the run-time layer a more accurate view, we instead share the bit vector with the operating system which is responsible for turning bits on or off as pages are transferred in or out of memory. Our actual implementation of the run-time layer and the shared data structure varies slightly in each of our experimental systems; we discuss these implementation details in Chapter 3.

2.3.3 The Operating System

To support compiler directed prefetching, the operating system needs to be able to respond to the prefetch and release requests issued by the application. We aim to make this support as simple as possible to avoid

adding unnecessary complexity to the operating system.

For a prefetch request, the operating system must perform actions similar to those required to handle a page fault. It allocates a free memory page to hold the requested data and issues an asynchronous read request to the file system before returning control to the application. When the data becomes available the page is mapped into the application's page table. In the event that there is no free memory available, the operating system is permitted to drop the prefetch request, rather than forcing pages to be evicted to make room for the prefetched data. This choice is reasonable because we have also added a mechanism to allow applications to limit their memory consumption by explicitly *releasing* pages that are not currently needed (although we do not depend on the use of this mechanism for correctness). If all pages are in use at some time, choosing to evict one to satisfy a prefetch request could hurt performance. The other major difference between handling a prefetch request and a normal page fault is that prefetches should not cause any memory access exceptions. If a prefetch attempts to read an address that the application is not permitted to access it should simply be discarded. This difference guarantees that the program will not terminate abnormally earlier than it would have without prefetching, and allows the compiler to occasionally generate incorrect prefetch addresses without causing any harm. This flexibility is extremely useful for the implementation of the compiler algorithm, since we do not need to be as careful when generating code at the limits of the arrays.

For a *release* request, the operating system performs actions similar to those of the paging daemon. It unmaps the specified page and places it on the free list, scheduling a write if necessary. Implementing both *prefetch* and *release* calls is relatively easy since they are similar to functions that the operating system must already provide for virtual memory management.

A more interesting issue is having the operating system actively share a data structure (i.e. the bit vector and memory usage summaries) with the user level. The operating system agrees to provide a user-level process with a range of memory that can be used as a map of the process's virtual memory usage. The operating system is responsible for allocating memory for the shared data, making it readable by the application, and mapping it into a specified location in the application's virtual address space. A new system call is thus needed to allow the application to request the shared data structure and provide the address that the applica-

tion will use to access it. The operating system must, of course, record the address of this structure for each prefetching application. Adding a field to the process structure would be a reasonable option.

As with the run-time layer, the actual implementation of the operating system support described here varies in each of our experimental systems, and the details are discussed in Chapter 3.

2.4 Evaluation Framework

The remaining chapters of this thesis present an evaluation of the basic design described in this chapter by examining the performance of a set of benchmark applications on two distinct implementations. Through this examination, limitations are identified and refinements to the basic design are introduced and evaluated. We therefore conclude this chapter with a presentation of the hardware platforms and their existing operating systems, and the benchmarks that will be used.

2.4.1 Hardware Platforms

Both of our hardware platforms are NUMA (non-uniform memory access) shared-memory multiprocessors; however, the specific memory model is unimportant to these experiments (all we require is support for virtual memory). Our choice of systems was motivated by support for high-bandwidth I/O subsystems and the availability of operating system source code and support from the authors of the operating systems. Despite the availability of source code, x86-based Linux systems were eliminated due to the lack of suitable I/O system support at the time the systems were chosen.

2.4.2 Research System Infrastructure

The first experimental platform used to evaluate our scheme is the Hurricane File System (HFS) [33] and HURRICANE operating system [56] running on the HECTOR NUMA shared-memory multiprocessor [60]. HURRICANE is a hierarchically clustered, micro-kernel based 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

Hardware Characteristics	
Processor	
Processor type:	Motorola 88100
Clock rate:	16.67 MHz
Data cache size:	16KBytes
Instruction cache size:	16 KBytes
Physical Memory	
Total size:	64 MBytes
Available to application:	48 MBytes
Page size:	4 KBytes
Disks	
Number of disks:	7
Maximum transfer rate:	640 KB/sec
Average rotational latency:	8.61 msec
Track-to-track seek time:	5 msec

Software Characteristics	
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

Table 2.2. HECTOR/HURRICANE characteristics

system policies applied to a file [33]. 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 2.2, and more detailed descriptions of the platform can be found in earlier publications [33, 56, 60]

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 chapters, the pages of the applications' data are striped by the file system round-robin across all seven disks. An extent-based policy is used to store the files on each of the disks, where contiguous file blocks are stored to contiguous blocks on the disk to avoid seek operations for sequential file accesses.

The 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.

2.4.3 Commercial System Infrastructure

To further validate our scheme for tolerating page fault latency in out-of-core applications, we also implemented our operating system support and run-time layer on a current commercial system. We used a

Processor	
Processor type:	MIPS R10000
Number of Processors:	4
Clock rate:	180 MHz
Physical Memory	
Total size:	128 MBytes
Available to application:	75 MBytes
Page size:	16 KBytes
Disks	
Manufacturer:	Seagate
Model:	Cheetah 4LP
Number of disks used for swap:	10
Maximum external (I/O) transfer rate:	40 Mbytes/sec/disk
Average rotational latency:	2.99 msec
Seek, read (min/max/avg):	0.98 / 18.2 / 7.7 msec (typical)
Seek, write (min/max/avg):	1.24 / 19.2 / 8.7 msec (typical)
Number of SCSI controllers:	5
Disks per controller:	2

Table 2.3. SGI Origin 200 characteristics.

Name	Description
BUK	integer bucket sort algorithm
CGM	solves an unstructured sparse linear system using the conjugate gradient method
EMBAR	monte-carlo simulation
FFT	3-D FFT PDE, performs forward and inverse FFT's
MGRID	computes 3-D scalar potential field on a uniform cubical grid using a multigrid solver
APPLU	solves four coupled parabolic elliptic PDE's using SSOR method to invert jacobian matrix
APPSP	solves five coupled parabolic elliptic PDE's using diagonalized approximate factorization method
APPBT	solves three coupled parabolic elliptic PDE's using b lock approximate factorization method

Table 2.4. Description of applications.

4-processor SGI Origin 200 [36], running our modified version of the IRIX 6.5 operating system to obtain our commercial system results. The system was configured so that approximately 75MB of physical memory was available to user programs, and the system swap space was striped across ten Seagate Cheetah 4LP disks using raw swap partitions. Five SCSI adapters each control two of these ten disks; the SCSI adapters are in turn connected to the PCI busses on the Origin. The basic hardware characteristics of our system are summarized in Table 2.3.

2.4.4 Benchmarks

To evaluate the effectiveness of our approach, we measured its impact on the performance of the entire NAS Parallel benchmark suite [6]. 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 is given in Table 2.4. 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 3.2.2. 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) [6]. 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 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.

Chapter 3

Impact on Out-of-core Applications

However beautiful the strategy, you should occasionally look at the results. — Winston Churchill

In this chapter we present and evaluate our implementation of the design outlined in Chapter 2. Our focus is on how effectively we are able to hide the latency of page faults to improve the performance of out-of-core applications. We therefore concentrate on the *prefetch* operation, and run all experiments with the machine dedicated to a single benchmark. Chapter 4 examines the effectiveness of the *release* operation for improving performance in multiprogrammed workloads.

We begin by describing our compiler algorithm in detail in Section 3.1, highlighting its evolution from a technique for prefetching data cache misses. This algorithm has been implemented as a pass in the SUIF (**S**tanford **U**niversity **I**ntermediate **F**ormat) compiler [26, 61]. The remaining two components of our system—the operating system support and the run-time layer—are tailored to match the existing infrastructure provided by the two systems described in Sections 2.4.2 and 2.4.3. We evaluate the basic system design in depth using the HURRICANE research platform in Section 3.2, then validate these results using the IRIX platform in Section 3.3. We conclude the chapter by discussing the limitations of our implementation and the lessons learned from our experimental evaluation.

	Cache Prefetching	I/O Prefetching
"Cache" Size	32 kB	$>$ 100 MB
"Line" Size	32 bytes	4 kB - 16 kB
"Miss" Latency	100 cycles	$>$ 100,000 cycles

Table 3.1. Comparing compiler input parameters for cache prefetching vs. I/O prefetching.

3.1 Compiler Algorithm

In this section 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 [39, 41]. 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, as shown in Table 3.1.

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

1. 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.

2. 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.

3. We need to generate *release* requests to identify pages that are no longer needed by the application.

We consider a *reference* to be an instruction that reads or writes a memory location. Specifically, since the compiler analysis is only applied to array accesses (i.e. `A[i]`), we will use the term *reference* to mean memory accesses through arrays.

Most of the required changes to the algorithm 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 3.1.1 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 [39] and is only reproduced here for completeness and ease of reference. Next, Section 3.1.2 discusses how the compiler first isolates these faulting instances through *loop splitting* techniques, and then schedules prefetches early enough using *software pipelining*.

3.1.1 Locality Analysis

The goal of the *locality analysis* step of the prefetching algorithm is to identify 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.

The key ideas needed for locality analysis are introduced first, followed by an illustrative example to show how these ideas can be expressed in terms of loop iterations. With this framework, we then describe each step of the locality analysis algorithm in detail.

Fundamental Concepts

A data item has *reuse* if it 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. Although most Unix-based operating systems employ some variant of a global least-recently-used (LRU) approximation for page replacement, many dynamic factors affect the choice of which pages will actually be evicted for a particular application. These factors may include the interaction with other applications, the relative size of each application's resident set, the size of the free page pool that the system tries to maintain, and the frequency with which pages are reclaimed. Since the actual replacement of pages at run-time is well beyond the compiler's ability to analyze statically, 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, and provides the most reasonable static approximation. 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.

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

(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

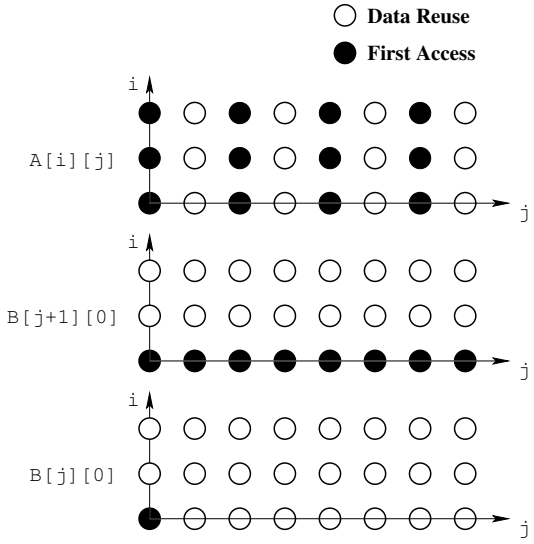


Figure 3.1. Data reuse example.

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 and the reuse analysis step, we now present a simple example.

An Example of the Reuse Vector Space

The types of reuse that can be identified using *reuse analysis* are shown in Figure 3.1(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 3.1(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.

The `A[i][j]` reference in Figure 3.1 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 3.1(b). This reference exhibits *spatial reuse*.

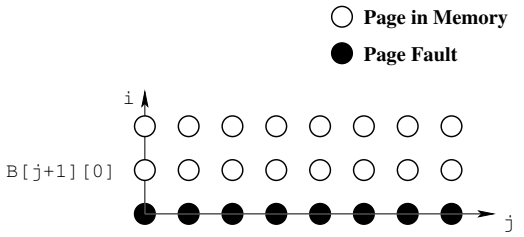
The `B[j+1][0]` reference traverses the columns of the matrix along the inner loop, causing each reference to touch a different page during the first iteration of the `j` loop. However, exactly the same locations are used again on subsequent iterations of the `i` loop, resulting in *temporal reuse* for this reference. This effect is illustrated in the second plot of Figure 3.1(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 3.1 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. We now describe how the compiler determines when reuse leads to locality, using the concept of the *localized iteration space*.

(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];
```

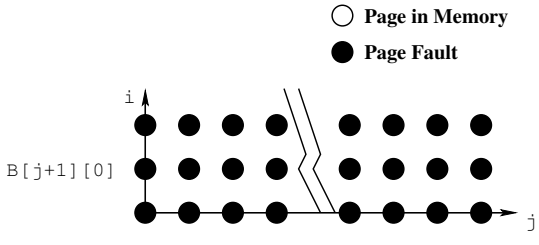


Figure 3.2. Example of how loop iteration counts affect locality.

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 3.2. 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 3.2(a) is identical to that in Figure 3.1, however in Figure 3.2(b) 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.

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 3.2(a) the `A[i][j]` reference brings $\frac{8*1}{2} = 4$ pages into memory, whereas in Figure 3.2(b) $\frac{10000*1}{2} = 5000$ pages are brought into memory. To find the number of 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 3.2, 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 3.2(a) and 10,000 for the code in Figure 3.2(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 3.2(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 iter-

ation 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. In contrast, for the code in Figure 3.2(b), a single iteration of the inner loop brings only three pages into memory (as before), 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.

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.:

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

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. In our implementation, we address this problem 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.

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

(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]`	$\begin{bmatrix} \texttt{i} \\ \texttt{j} \end{bmatrix} = \begin{bmatrix} \text{none} \\ \text{spatial} \end{bmatrix}$	(`j` mod 2) = 0
`B[j+1][0]`	$\begin{bmatrix} \texttt{i} \\ \texttt{j} \end{bmatrix} = \begin{bmatrix} \text{temporal} \\ \text{none} \end{bmatrix}$	`i` = 0
`B[j][0]`	$\begin{pmatrix} \text{Group with} \\ \texttt{B[j+1][0]} \end{pmatrix}$	*False*

Figure 3.3. Example of how prefetch predicates are constructed.

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` = `initial_value`". 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 3.3(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 3.3.

The reasoning applied to the locality vector space representation to construct prefetch 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. There are some additional considerations which complicate the problem of scheduling releases at the right time, which we discuss in detail in Chapter 4.

3.1.2 Scheduling Prefetches

Most of the changes made to the original compiler algorithm for prefetching [39] 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 empha-

(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 3.4. Example of peeling the first and last iterations of a loop

(a) Original Code

```
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 Unrolling

```
for (i = 0; i < n; i+=64) {
  f(i);
  g(i);
  g(i+1);
  g(i+2);
  ...
  g(i+63);
  h(i);
}
```

(c) 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 3.5. Example of unrolling and strip-mining a loop by a factor of 64

size the changes required for I/O prefetching.

Loop Splitting Techniques

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

The simplest case is 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 on a generic loop is shown in Figure 3.4.

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 or thousand elements will fit on a page), and the preferred technique is to *strip-mine* the loop by a factor of l, since replicating the loop body several hundred times is unreasonable. Examples of both unrolling and strip-mining are shown in Figure 3.5. These large strip-mining factors lead to one additional complication: if a given loop nest accesses less than a full page of data, it will be impossible to strip-mine the loop. Since many items fit on a page, it is quite likely that the innermost loop, and even several surrounding loops, will not access enough data to make strip-mining feasible. In contrast, with cache prefetching it was extremely unlikely that a loop would access less data than a cache line, thus it was reasonable to focus on the innermost loop. 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 expansion 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.

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* [35]. A simple example of this technique is shown in Figure 3.6. 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 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 3.1.2 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 of the reference 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.

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:

$$d = \left\lceil \frac{l_m}{s + l_p} \right\rceil \tag{3.1}$$

(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 3.6. 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.

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 3.6 shows a simple example of how prefetches and releases are scheduled using software pipelining. For this example, we have chosen parameters that correspond to the HURRICANE 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 3.1, the compiler then determines that six iterations of the outer `i1` loop (with a prefetch call added) are needed to hide the I/O latency completely. 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 3.7(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 four pages with each prefetch. Since each iteration of the `m` loop accesses only 20 bytes of data, it would take 820 iterations of the `m` loop to cover a four page block, which means that strip-mining is not performed since only five iterations are available. Using equation 3.1, the compiler determines that prefetches need to be issued roughly 12,000 iterations of the `m` loop ahead of the reference, causing a block prefetch for 12 pages to be inserted for the prolog section of the pipeline, as shown in Figure 3.7(b). Now, since the spatial locality of the `A[i][j][k][l][m]` reference suggests that we only need to prefetch every 820 iterations of the `m` loop, 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

(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 3.7. Example of Software Pipelining with Small Loop Bounds.

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 3.7(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 Indirect references such as `A[index[i]]` are assumed to have no locality, hence 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 reference itself (`A[index[i]]`), 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 [39].

A More Detailed Example Figure 3.8 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 3.8(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

(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 3.8. 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.)

prefetch and release requests together whenever appropriate, to minimize the number of system calls.

3.1.3 Compiler Implementation

We implemented our prefetching algorithm as a pass in the SUIF (Stanford University Intermediate Format) compiler [26, 61]. The output from the SUIF compiler is C code containing prefetch and release calls (as illustrated in Figure 3.8(b)). We also use the SUIF compiler to convert the original Fortran source code of each benchmark into C code for the original, non-prefetching versions that we use in our experiments. We then compile the resulting C code into an executable for our target systems using gcc version 2.5.8 (with the

-O2 optimization flag set) for the HURRICANE system and SGI's MIPSpro compilers version 7.2.1 (with the -O3 optimization flag set) for the IRIX system. 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 the bit vector to filter prefetch requests are inlined for performance.

3.2 An Initial Prototype: HURRICANE

The HURRICANE operating system was designed with a micro-kernel structure. A small number of vital services, such as memory management, are provided by kernel-level servers. The Hurricane File System (HFS), and the majority of other operating system services are provided by user-level servers, however. A very fast inter-process communication mechanism, called *Protected Procedure Calls* (PPC) [23], are used for most communication between user programs, kernel servers, and user-level servers. One important feature of HURRICANE is that it was designed for multiprocessors and has strong support for concurrency. Each server maintains a pool of light-weight worker processes which are used to handle client requests. We take advantage of this feature to implement asynchronous prefetch requests—control can be returned to the application as soon as a worker thread hands the request to the file system.

Because HURRICANE is a research operating system, it does not contain all the functionality that is required of a commercial system. In particular, it does not have support for writing and reading virtual memory pages to and from swap space. Instead, we needed to modify the NAS Parallel benchmark programs to use mapped files to provide the backing storage space for their data in the HURRICANE experiments.

The overall structure of HURRICANE, and the implementation of our scheme on this platform, is illustrated in Figure 3.9. We now take a detailed look at this implementation.

3.2.1 Operating System and Run-Time Layer Implementation

HURRICANE supports our approach in the following three ways. First, when a prefetch request is received by the operating system, the memory manager 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, a worker thread is allocated to send

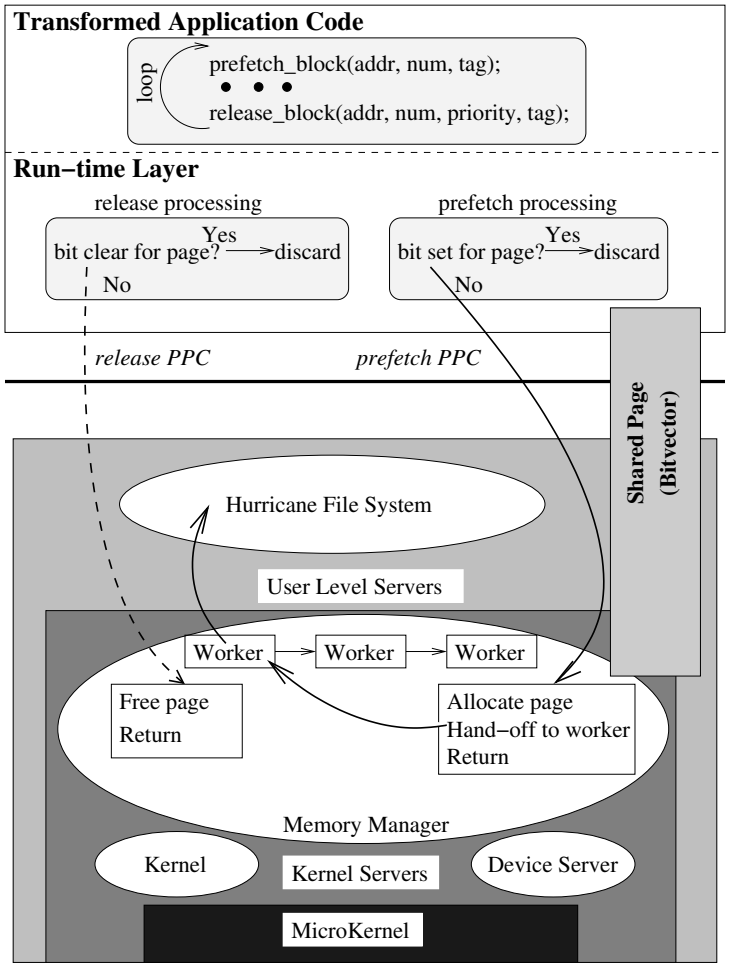


Figure 3.9. Implementation of prefetching and releasing support on HURRICANE.

an asynchronous read request 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.

When a release request is received, the memory manager simply 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. HURRICANE uses a separate system daemon to periodically scan the free list and schedule writes for any dirty pages, so our implementation does not need to check for this case separately.

The shared bit vector is implemented using a single 4 kB page of physical memory, which is used as a map of the process's virtual address space. HURRICANE uses physical addresses to access the shared page, so a larger bit vector would require either (i) the allocation of multiple contiguous physical pages, or (ii) a more complex strategy for accessing the bit vector each time a bit needs to be updated. Limiting the bit vector to a single page provides simplicity and fast access. However, with a 4 kB page size and each bit representing a single page of the virtual address space, the bit vector is capable of tracking only 2^{15} pages of virtual memory or 134 MB of data. This is an unreasonably small amount of data for "out-of-core" applications; our solution on HURRICANE is to allow each bit to represent one or more contiguous virtual memory pages. We refer to the number of pages represented by each bit as the *granularity* of the bit vector.

If the application accesses more than 134 MB of data, 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 implementation the decision is made by the run-time layer. 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 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 corresponding bit is turned on, 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 turns bits on 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 turned on. In turn, when a single page is brought into memory due to a page fault the operating system does not turn the bit on, allowing the run-time layer to still issue prefetch requests for the other pages in the group.

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. Information about the size of mapped files is used to set the granularity of 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 turn bits on in the vector whenever the process page faults or prefetches, and clear them whenever pages belonging to the process are unmapped.

During execution of the modified application, the run-time layer uses the shared bit vector to *filter* prefetch and release requests, discarding them if no further action is necessary. For block prefetch requests, the run-time layer checks the bit for each page in the block 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, a request is issued to the operating system for the missing page and all subsequent pages in the block. This strategy ensures that at most one system call is required for a block prefetch. In the operating system, worker threads are used to allow each page in a block request to be fetched in parallel.

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

Name	Input Data Set	Memory Required		Original Execution Time (mins)
		Absolute	% of Available	
BUK	2^{23} 19-bit integers	103 MB	215%	21.0
CGM	sparse matrix with 7,607,024 non-zeros	103 MB	215%	57.2
EMBAR	2^{24} random numbers	134 MB	279%	53.9
FFT	128x128x128 matrix of complex numbers	117 MB	244%	87.9
MGRID	128x128x128 matrix	58 MB	121%	31.9
APPLU	5x5x64x64x32 matrices	120 MB	250%	48.9
APPSP	90x90x90 matrices	117 MB	244%	224.3
APPBT	5x5x64x64x32 matrices	94 MB	196%	85.2

Table 3.2. Application characteristics on HURRICANE.

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 the HURRICANE results section (Section 3.2.2).

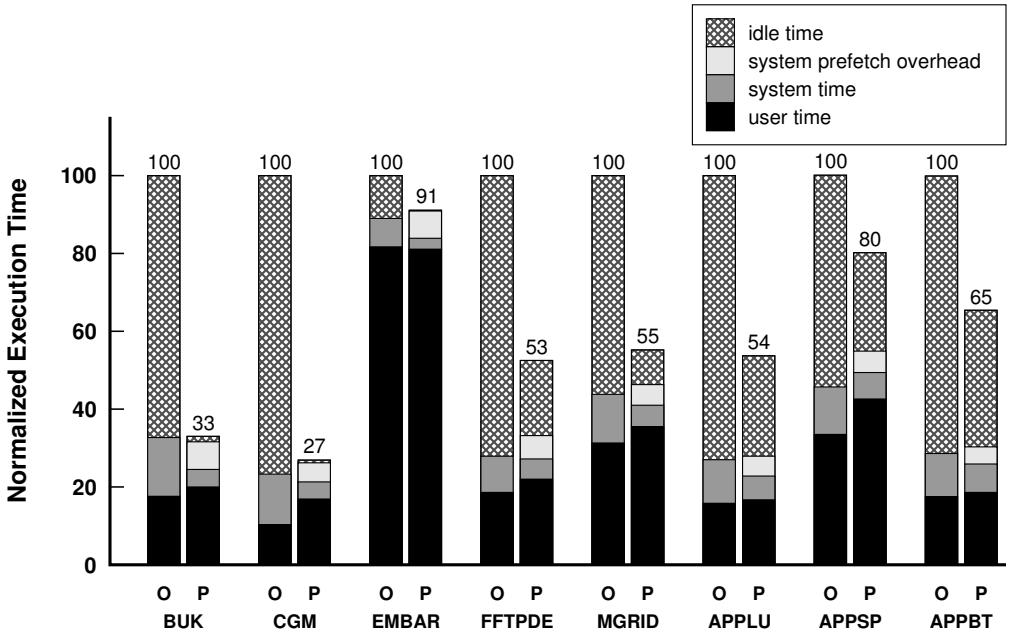
3.2.2 Evaluation of HURRICANE Implementation

We begin by focusing on the impact of our scheme on overall execution time. We then look at the performance from a system-level perspective, including the effects on disk and memory utilization. Next, we evaluate the effectiveness of the compiler and the run-time layer at scheduling prefetches and reducing overhead. To assess the robustness of our results, we then study the effect of varying problem sizes. Finally, we presents a detailed case study of one of the benchmark applications, BUK.

Overall Performance

Table 3.2 gives basic characteristics of the NAS Parallel benchmarks as executed on the HURRICANE platform, including a description of the data set, the total amount of memory required (both in megabytes and as a percentage of the physical memory available), and the absolute time required to execute the original non-prefetching versions.

Figure 3.10(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



(a) Execution Time Breakdown (**O** = original, **P** = with prefetch)

Benchmark	Original		With Prefetch		
	Total Faults (x1000)	Avg. Stall Time (msec)	Total Faults (x1000)	Avg. Stall Time (msec)	I/O Stall Reduction (%)
BUK	41.529	24.5	0.810	16.1	98.7%
CGM	135.066	22.0	0.207	26.5	99.8%
EMBAR	65.535	7.7	0.005	13.5	100.0%
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 3.10. Overall performance improvement from prefetching on HURRICANE.

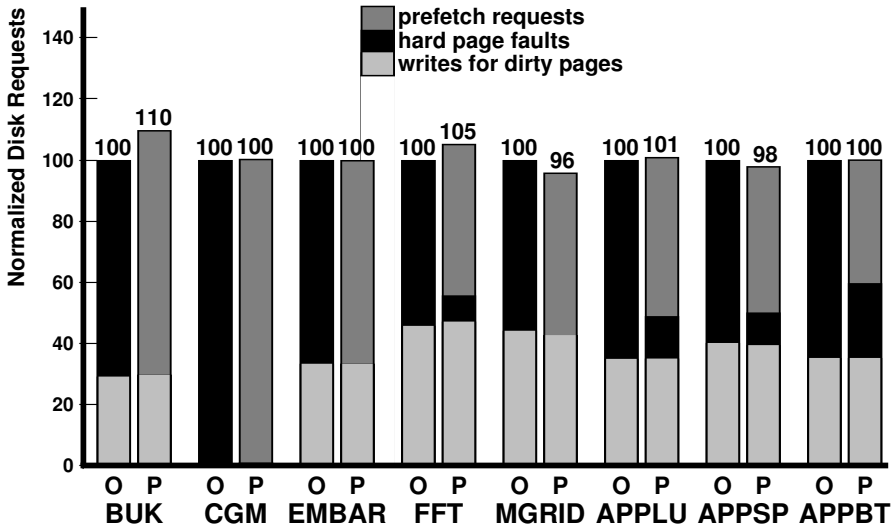
prefetch and release data explicitly (**P**). In each bar, the top section is the amount of time when the processor was idle, which corresponds roughly to the I/O stall time since we run only a single application during these experiments. The bottom section of each bar is the time spent executing in 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 3.10(a), the speedup in overall performance ranges from 9% to 270%, with the majority of applications speeding up by more than 80%. Figure 3.10(b) presents additional information on page faults[1] and stall time. As we see in Figure 3.10(b), more than half of the I/O stall time has been eliminated in seven of the eight applications, with three applications eliminating over 98% of their I/O stall time.

Having established the benefits of our scheme, we now focus on the costs. Figure 3.10(a) shows that the instruction overhead of generating prefetch addresses and checking whether they are necessary in the run-time layer causes less than a 20% increase in user time in five of the eight applications—in the worst case (CGM), the user time increases by 70%. However, in all cases this increase is quite small relative to the reduction in I/O stall time. If we focus on the system-level overhead of performing prefetch operations, we see in Figure 3.10(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.

[1] 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) 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			
Bench	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 3.11. Impact of prefetch and release on system resources on HURRICANE.

Disk and Memory Utilization

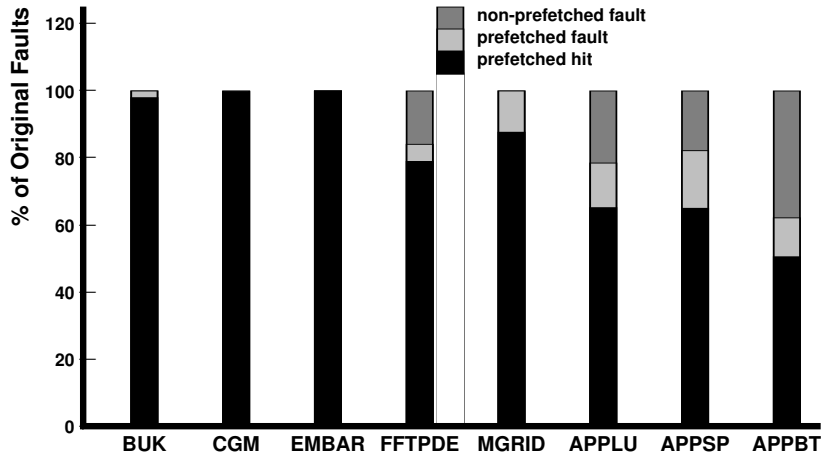
In Figure 3.11(a) and (b) we break down the types of requests seen by the disks and show average disk utilization during execution for both the original and prefetching versions of the applications. In almost all cases, the total disk requests do not increase as a result of prefetching, and for two of 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 3.11(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 3.11(c). Since our initial compiler implementation was not aggressive about inserting release operations, most applications did 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 their working sets are significantly smaller than their total data sets. We did not find that releases had a significant impact on out-of-core application performance on a dedicated machine in the HURRICANE environment. Chapter 4, however, examines the benefits of releases on a multiprogrammed workload in the IRIX environment.

Effectiveness of the Compiler and Run-Time Layer

Figure 3.12 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.

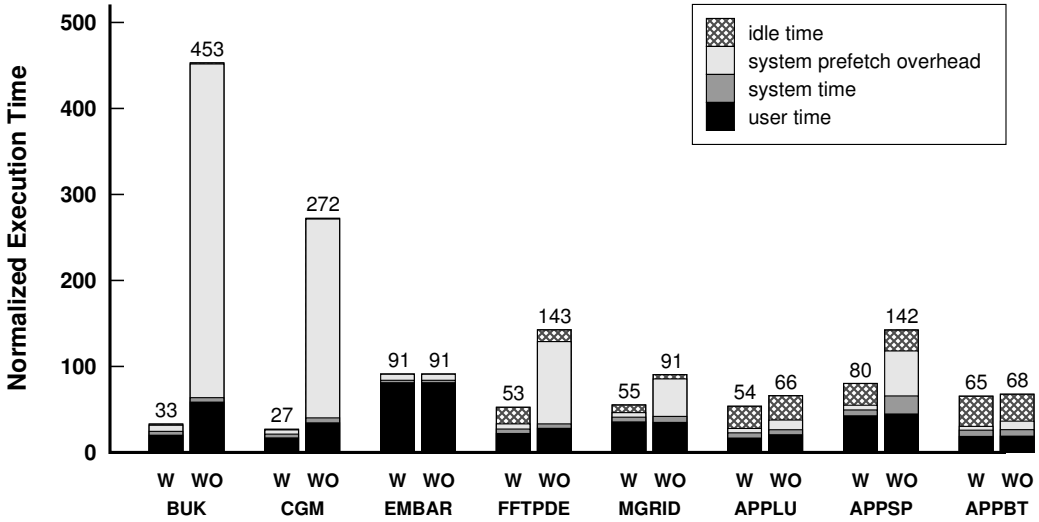
The Compiler To assess the compiler analysis, we consider what happens to the original page faults when prefetching is added. There are three possibilities: (i) a previously faulting page is successfully prefetched (we call this a *prefetched hit*), (ii) a faulting page is prefetched but still faults 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*).



(a) Impact of prefetching on the original page faults.

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 3.12. Effectiveness of the compiler analysis and run-time filtering.

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 3.12(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. Briefly stated, they are:

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

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 3.1.2). In Chapter 5 we develop and evaluate a new algorithm for scheduling prefetches in the presence of unknown loop bounds that overcomes some of these problems.

The second instance in which some pages are not prefetched arises when the data is not well-aligned with respect to the page boundaries. When calculating group reuse, our compiler implementation assumes that two references will probably touch the same page if the data locations that they access are separated by less

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

Figure 3.13. Example of a reference not recognized as an array reference (inside `foo`) by the compiler.

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 3.13. 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 only recognizes the `B[i]` reference. The reason is that earlier passes of the SUIF compiler have converted the high-level source code shown in Figure 3.13 to an internal representation where `&A[i]` is not identified as an array reference.[2] 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 3.12(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 3.12(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

[2] 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`.

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

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

Figure 3.14. Example of reuse not identified by the compiler.

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 3.1.2, the result is that prefetches are only issued in the prolog and the steady state is never reached. Again, our solution to this problem is presented in Chapter 5

Finally, the middle column of Figure 3.12(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. Locality analysis is not applied to these types of references, instead, the compiler assumes that each such reference could touch a different page of data. For these applications this "worst-case" behavior rarely occurs and most of the prefetches are unnecessary. We will examine the utility of prefetching indirect references more closely during our case-study of BUK. In all cases, unnecessary prefetches occur whenever the compiler underestimates memory's ability to retain data. One cause of this effect is that locality analysis is applied to each set of nested loops independently—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. Thus, the type of reuse shown in Figure 3.14 cannot be discovered by the current compiler algorithm. 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. All array references are sequential and within one-dimensional loops, thus the problems of strip-mining the loop correctly and choosing a pipelining loop (discussed in Section 3.1.2) do not occur. Furthermore, since the single array used in EMBAR is large enough to flush all of memory, there are no unnecessary prefetches that result from only considering a single loop at a time.

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 3.12(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 3.12(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 3.12(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 3.12(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 3.10(a) increases significantly for some applications.

Figure 3.12(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.

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

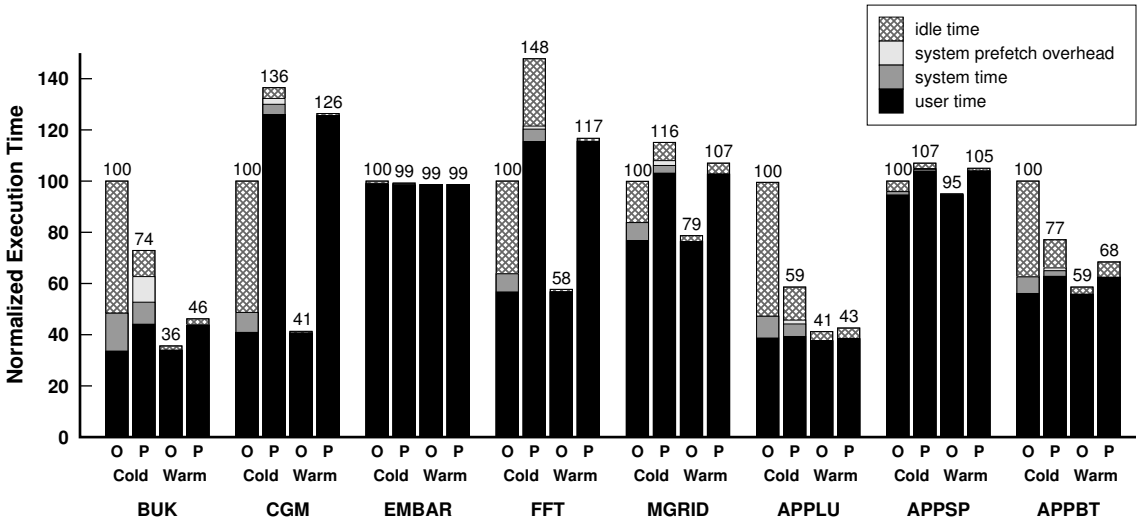


Figure 3.15. Performance with in-core data sets (O = original, P = with prefetch; Cold = cold-started, Warm = warm-started). Performance is normalized to the original, cold-started cases.

or no benefit. Figure 3.15 shows two sets of experiments—the cold-started and warm-started cases—on data sets that are roughly 10-35% as large as the available memory. Starting with the cold-started cases, we see that prefetching degrades performance in four cases, but actually *improves* performance in three cases (BUK, APPLU, and APPBT) by hiding the latency of cold page faults. To further isolate the prefetching overhead, we also warm-started the applications by preloading all of their data from the input files into memory before timing the runs. As expected, prefetching typically degrades performance in the warm-started cases since it offers no potential advantage. However, we believe that the cold-started cases are more realistic for most applications, since real programs must read their input data from disk.

In these experiments, we made no attempt to minimize prefetching overhead for in-core data sets, but this is a problem that we are planning to 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 3.16 shows the performance of

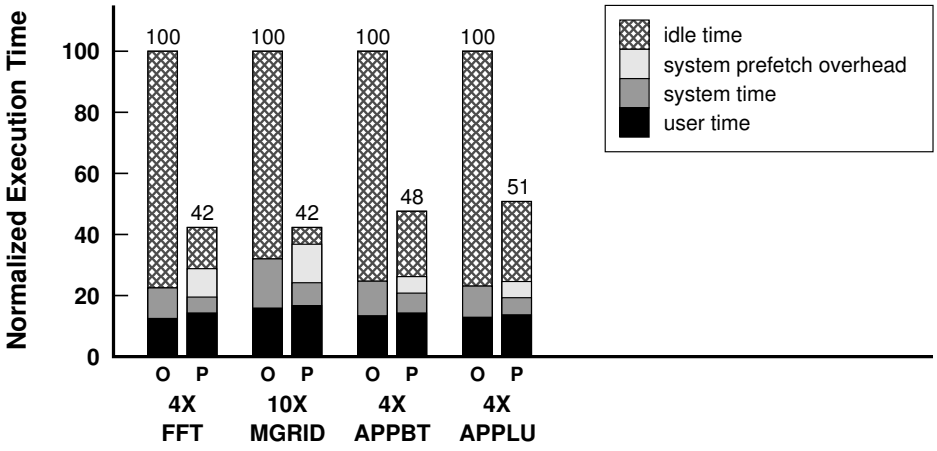


Figure 3.16. Performance with larger out-of-core problem sizes. Numbers above application names indicate how much larger the problem sizes are than available memory.

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 3.2.1 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 3.16) requires 464 MB of memory, which is approximately 10 times more than what is available, and each bit in the bit vector must represent 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 3.16, 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 earlier in this section.

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 3.17 shows the main computations performed by BUK as C source code. (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.) 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 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, which we will refer to as the *copying phase*, `rank` and `key` are accessed by direct references while `key2` is accessed indirectly.

All of the array references in BUK are identified and prefetched by our compiler (the bar for BUK in Figure 3.12(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

```
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 3.17. Source code (C representation) for BUK

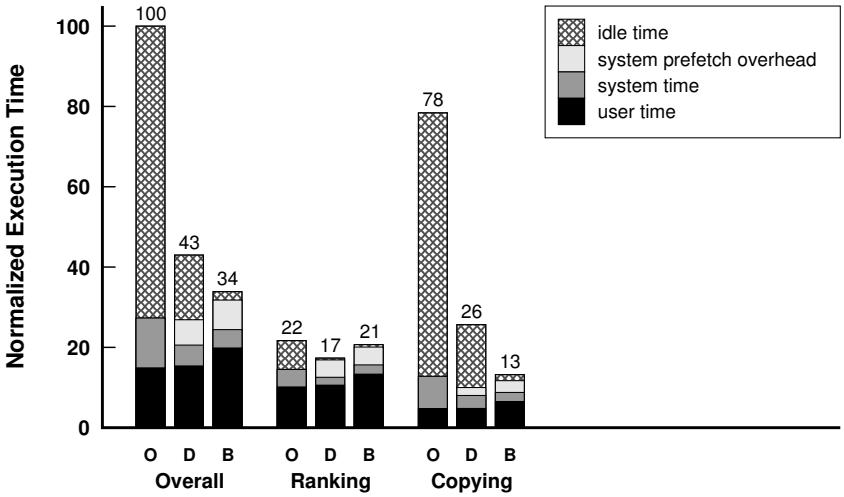


Figure 3.18. Performance for different phases of BUK, normalized to the original, non-prefetching case (O = original, D = with direct-only prefetching, B = with direct and indirect prefetching).

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 3.18. In this figure, all bars have been normalized to the original, non-prefetching execution time. The first set of three bars (labeled **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 phase) and issue indirect prefetches aggressively when the locality is poor (as in the copying phase). This could potentially be accomplished by

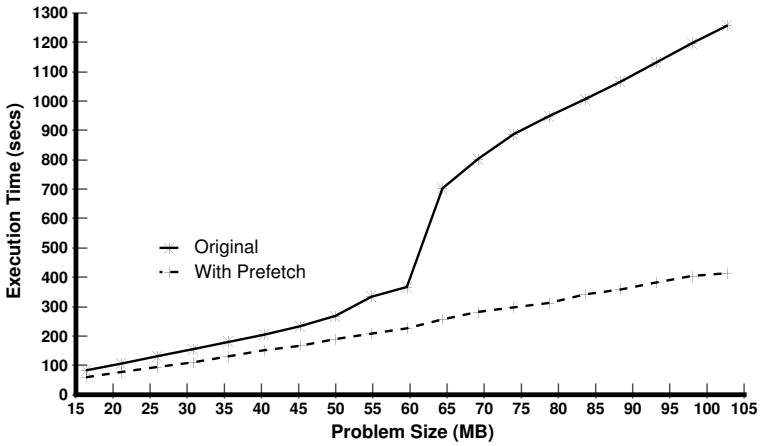


Figure 3.19. Performance of BUK (cold-started) across a range of problem sizes.

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 3.19.

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, with about 48 MB 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 consistently outperforms the original code, since even small problem sizes benefit from prefetching cold misses. (For BUK, it is more realistic to cold-start the application, since it must always read its input data set from disk.) Hence this application exemplifies what we are attempting to accomplish with automatic I/O prefetching: programmers can write their code in a natural manner and still achieve good performance, even for out-of-core data sets.

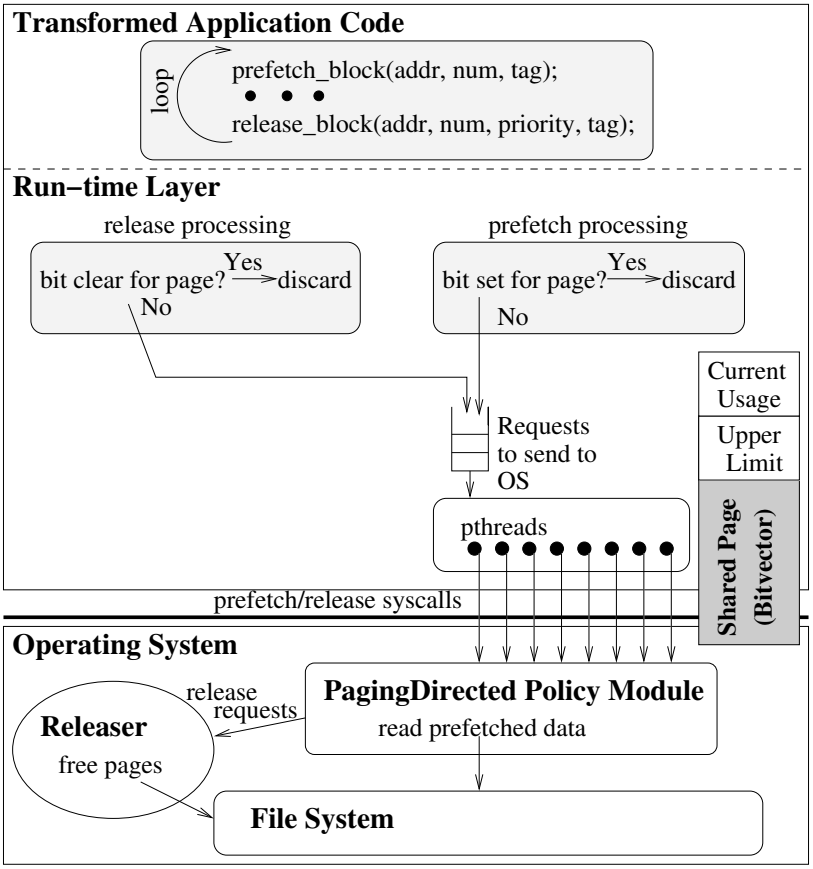


Figure 3.20. Implementation of prefetching and releasing support on IRIX.

3.3 Experience with a commercial operating system: IRIX

Like HURRICANE, IRIX 6.5 is designed to run on high-performance multiprocessors, however unlike HURRICANE, it does not have a microkernel design. Major subsystems in IRIX include scheduling, interprocess communication, memory management and file systems. The overall structure of our implementation on this platform is illustrated in Figure 3.20. We now take a detailed look at this implementation, focusing again on the prefetch operation and its performance impact.

3.3.1 Implementation

We have implemented support for user-level paging directives (i.e. prefetch and release) within the SGI IRIX 6.5 operating system. IRIX 6.5 supports a *Memory Management Control Interface*, which consists of policy modules that allow users to select various policies for page size, allocation, migration, and

replication. A policy module may be connected to any range of an application's virtual address space, down to the level of a single page. We have defined a new policy module—called "*PagingDirected*"—that allows a user-level process to invoke prefetch and release operations on pages of its address space associated with this policy. In addition, the *PagingDirected* policy module shares information about memory usage with the application through a single 16KB page. This page is allocated by the operating system and mapped read-only into the application's address space when the *PagingDirected* policy module is created. The page is used primarily as a bitmap, indexed by virtual page number, in which bits are turned on to indicate that the corresponding page is in memory, and cleared otherwise. We also use the shared page to convey additional information to the run-time layer, such as the amount of memory still available and the current size of the application's resident set. We will examine how the run-time layer uses this information in Chapter 4.

When the *PagingDirected* policy module receives a request to prefetch a page, it performs actions similar to those that occur for a page fault, with two notable exceptions. First, if there is no free memory available to allocate for the prefetched data, the prefetch request is discarded immediately. Second, when the request completes, the prefetched page is not fully validated and no entry is made in the TLB. The second feature prevents mappings for prefetched (and not yet referenced) pages from displacing TLB entries which are still in use.

Requests to release pages are handled by passing the released addresses to a new system releasing daemon—called the releaser—which is similar in function to the paging daemon, but is specialized to reclaim only the pages specified by the application. When a release request is made, the *PagingDirected* policy module clears the bits for the pages and enters the request in the releaser's work queue. The releaser handles requests from each prefetching/releasing application as they are received, first checking the bit vector to make sure that the pages have not been referenced again (either by a prefetch or a real reference) between the time that the application made the request and the time that the request is handled. The releaser then performs all actions needed to free the pages, including the allocation of swap space and writing back dirty pages if necessary. Released pages are placed at the end of the free list, so that they will not be reallocated for another purpose immediately. This strategy gives pages that were released too early a chance to be rescued from the

free list.

All updates to the shared page are handled by the operating system. When the *PagingDirected* policy module is created, all bits in the shared page are initially set. When the application attaches the policy module to a region of its virtual address space, the bits corresponding to those addresses are all cleared. Thereafter, bits are turned on whenever a physical page is allocated for a virtual page associated with this policy module, either due to prefetch requests or ordinary page faults. Bits are cleared when pages are reclaimed, either by an explicit release request or due to default page replacement activity. Note that since the base page size in IRIX 6.5 is 16KB, we are able to represent 2GB of memory using a granularity of one page per bit, which is sufficient for a 32-bit address space. For 64-bit address spaces that are expected to be sparsely populated, a multi-level bit vector scheme may be more appropriate than requiring a single bit to represent multiple pages.

To achieve the full benefit of prefetching, we need to be able to both fetch data asynchronously (so the application can continue after issuing the prefetch) and take advantage of any available parallelism in the underlying disk subsystem. The run-time layer accomplishes these requirements by creating a number of *Pthreads* [29] that make the actual calls to the *PagingDirected* policy module and wait for the prefetches to complete. When a prefetch request inserted by the compiler is intercepted by the run-time layer, the bitvector is first checked to see if a prefetch is really needed. Then, if necessary, the request is placed on a work queue and one of the prefetching threads is signaled to handle the request. The prefetching threads simply remove requests from the queue and issue them to the *PagingDirected* policy. This Pthreads-based approach to achieving asynchronous prefetching is very similar to the implementation of the asynchronous I/O library in IRIX.

As with HURRICANE, extensive instrumentation was added to the IRIX memory management routines to enable us to evaluate our approach.

3.3.2 Evaluation of IRIX Implementation

Having demonstrated the effectiveness of compiler-inserted I/O prefetching on a research platform, we now focus on whether these significant performance gains can also be achieved on a modern commercial

Name	Input Data Set	Memory Required		Original Execution Time (mins)
		Absolute	% of Available	
BUK	2^{24} 20-bit integers	206 MB	275%	9.5
CGM	sparse matrix with 15,167,342 non-zeros	206 MB	275%	49.2
EMBAR	2^{24} random numbers	134 MB	179%	8.4
FFT	256x128x128 matrix of complex numbers	235 MB	313%	28.3
MGRID	256x256x256 matrix	452 MB	600%	14.9
APPBT	5x5x64x64x64 matrices	189 MB	252%	37.3
APPLU	5x5x62x62x62 matrices	219 MB	292%	13.6
APPSP	110x110x110 matrices	213 MB	284%	74.9

Table 3.3. Application characteristics on IRIX.

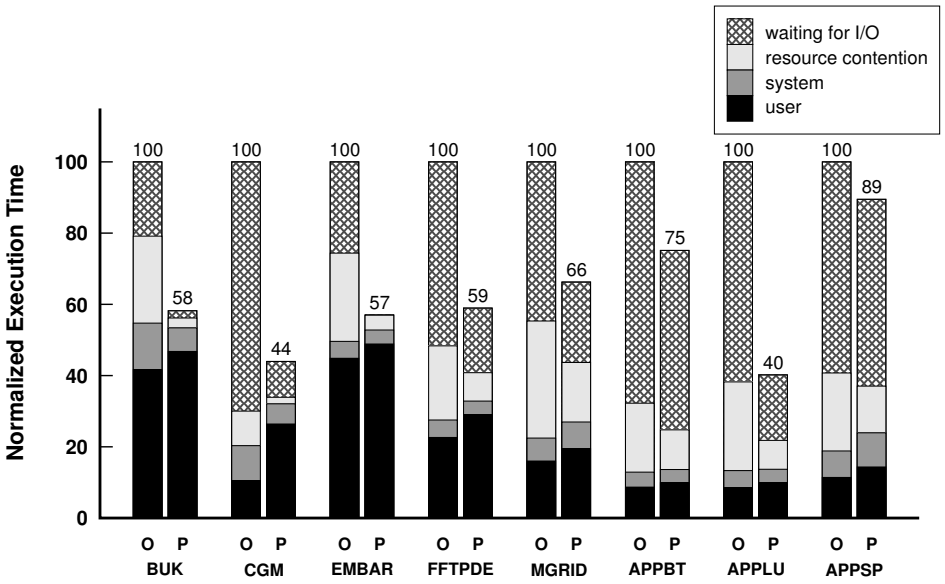


Figure 3.21. Overall performance improvement from prefetching and releasing on IRIX (O = original, P = with prefetching and releasing).

system—in this case, an SGI Origin 200 machine [36] running our modified version of IRIX 6.5. Since this modern system has more available physical memory than the research platform we considered earlier (75 MB vs. 48 MB, as discussed earlier in Section 2.4.1), we have increased the problem sizes of the NAS Parallel benchmarks accordingly, as shown in Table 3.3.

Figure 3.21 shows the results of our experiments, where execution time is once again broken down into four categories. Just as before, the top section is I/O stall time, the bottom section is user mode time, and the middle two sections are system mode time. In contrast with our HURRICANE experiments, however, these latter two components are now broken down into time spent executing system code (*system*), which in this

case is primarily time spent in the fault handling code, and time spent waiting for resources held by other processes (*resource contention*) such as locks, the CPU, memory, etc.

As we see in Figure 3.21, most of the applications are enjoying large performance gains as a result of compiler-inserted I/O prefetching on this commercial system. In six of the eight cases (BUK, CGM, EMBAR, FFT, MGRID, and APPLU), the I/O stall times have been reduced by 50% to 99%, thus resulting in overall program speedups ranging from 34% to over twofold. In the other two cases (APPSP and APPBT), I/O stall times are reduced by only 11% to 25%, resulting in more modest overall program speedups. While a direct comparison with the earlier HURRICANE experiments would be meaningless because so many parameters have changed (e.g., the hardware, the system software, the application inputs, etc.), we nevertheless observe the same general trends. Note that APPLU shows much greater benefit on the IRIX system, however. The poor performance on HURRICANE was the result of a default compiler option that limited code growth and prevented the prefetching transformation from being applied in one critical routine. This limit was increased for the IRIX experiments, allowing us to obtain a better indication of APPLU behavior with prefetching.

In all cases, we observe in Figure 3.21 that system overheads (i.e. *system time* and *resource contention* combined) actually *decrease* once we add prefetching and releasing. There are two reasons for this. First, prefetch requests are serviced by separate threads (implemented using *Pthreads* [29], as discussed earlier in Section 3.3) that can potentially run on other processors, since the Origin 200 is a multiprocessor. Hence some of the system software overhead associated with servicing page faults can potentially be overlapped with useful computation. Second, by using release operations to keep a sufficient amount of physical memory free, we can avoid resource contention with the system paging daemon as it tries to determine which pages it should reclaim. (A detailed analysis of this latter effect was published recently [10].) While the reduction in I/O stall time is the dominant effect in improving performance, many applications also benefit significantly from these overhead reductions.

To help gain further insight into the performance of these applications on IRIX, Figure 3.22 shows a breakdown of how prefetching affected the original page faults. It is interesting to compare this graph with Figure 3.12(a), which showed the same breakdown for the HURRICANE experiments. As expected, the frac-

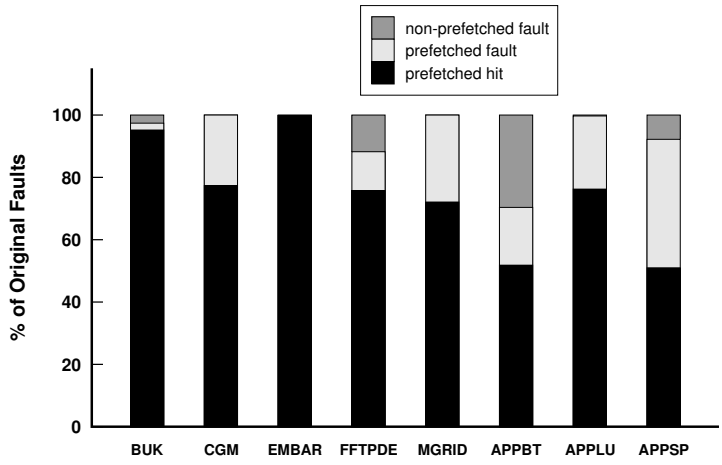


Figure 3.22. Impact of prefetching on the original page faults under IRIX.

tion of original page faults that the compiler failed to prefetch (i.e. the *non prefetched fault* category) is quite similar across the two systems, since it largely reflects limitations in the compiler analysis that are common across both platforms (with the exception of APPLU, where prefetching on HURRICANE was restricted due to code growth limits as described earlier). The most noticeable change between the HURRICANE and IRIX experiments is the relative fraction of prefetches that were early enough (i.e. *prefetched hit*) versus too late (i.e. *prefetched fault*). Comparing Figure 3.22 with Figure 3.12(a), we see that more of the prefetches on the IRIX platform were not launched early enough, and thus failed to hide all of the page fault latency. The reason is that we have a significantly larger relative disk latency on the Origin 200, due to its much faster processors, which is harder to hide fully. Problems with late prefetches in the HURRICANE experiments that were already apparent in Figure 3.12(a)—as discussed in Section 3.2.2—tend to be amplified, and are exposed in new places. The most dramatic example of this effect is CGM: roughly 23% of the prefetches are issued too late under IRIX, while they were nearly perfect under HURRICANE. To overcome this limitation, the compiler must do a better job of using software pipelining to schedule prefetches early enough in the presence of small or statically unknown loop bounds. Despite these late prefetches, however, we are still achieving impressive performance gains for the majority of the applications, confirming that compiler-inserted I/O prefetching is an effective technique for accelerating out-of-core applications, even on state-of-the-art commercial systems.

3.4 Lessons and Limitations

From our experiences with the two implementations of our design for compiler-directed I/O prefetching described in this chapter, we extract the following lessons:

- The overall design and distribution of responsibilities is effective. Sharing information between the system components enables better decisions than would be possible using the information available to any single component in isolation.
- A user-level dynamic adaptation layer is critical for achieving good performance. Although very effective in many cases, the compiler analysis will never achieve perfect static prediction, and it is too expensive to consult the operating system to resolve each mistake.
- Complicated operating system memory management policies are not required. Simple support allowing applications to specify their own needs, combined with exposing information to the user-level, is sufficient.

In the cases where our system failed to hide the I/O latency completely, we have repeatedly found that the compiler analysis is hampered by the need to choose a single code transformation with incomplete information (due to symbolic loop bounds, varying memory availability and other compile-time unknowns). While dynamic adaptation is a key component of our system, our run-time layer is only able to react to requests inserted by the compiler. In the case of prefetches, if the compiler fails to insert a request for a given reference, or fails to schedule the request early enough, the run-time layer cannot correct the situation. To address this limitation, the compiler needs to generate code so as to effectively defer the scheduling of prefetches until run-time, which is the subject of Chapter 5. In the case of releases, the compiler may insert a request for a given reference too early. We address this in Chapter 4, where we describe extensions to the compiler algorithm which allow the run-time layer to buffer release requests until they are needed, and then prioritize them based on their inherent reuse characteristics.

Chapter 4

Performance in Multiprogrammed Environments

Prediction is very difficult, especially about the future. — Niels Bohr

In the first part of our evaluation (Chapter 3), our focus was on using *prefetching* to hide the *I/O latency* of out-of-core applications running on a *dedicated* machine. In this chapter, we focus on using *release* operations to *manage physical memory* intelligently within a *multiprogramming workload* that includes an out-of-core application. This problem is important for three reasons. First, we may be able to leverage our knowledge of access patterns (extracted via compiler analysis) to implement application-specific replacement policies. By managing main memory more efficiently, demand on the I/O subsystem can be reduced, allowing prefetching to be more effective. Second, we can reduce the amount of work performed by the page daemon by explicitly identifying pages that can be freed. By replacing work done by the paging daemon at *run-time* with work done at *compile-time*, we can spend more time executing user programs instead of the page daemon. Third, we can limit the amount of memory consumed by the out-of-core program, leaving more memory pages free for use by other applications.

Although we included the release operations for the results in Chapter 3, we did not attempt to improve

on the compiler analysis dynamically, beyond filtering out releases for pages that were not in memory. On the HURRICANE platform, for stand-alone out-of-core applications, explicitly releasing memory did not result in a significant performance benefit over prefetching alone. The benefits of releases on the IRIX platform are quite different, and we explore them in detail in this chapter.

We begin our discussion in Section 4.1 by examining why out-of-core programs cause problems for the default memory management policies of operating systems, especially when mixed with interactive tasks that have very different characteristics. We then briefly consider some alternatives for addressing this problem in Section 4.2, and advocate a pro-active approach in which out-of-core programs are given the ability to control their own memory usage. Details of our compiler algorithm for explicitly releasing pages are given in Section 4.3 and the extensions to the operating system and run-time layer in Sections 4.4 and 4.5, respectively. The impact of the release operation on the out-of-core and interactive application performance is the subject of Section 4.6; out-of-core performance is studied in Section 4.6.2, the effectiveness of the release operation is studied in Section 4.6.3 and the effect on concurrently executing interactive applications is examined in Section 4.6.4. We end the chapter by summarizing the main results in Section 4.7.

4.1 Issues with Interactive Applications

In the previous chapter, we demonstrated that out-of-core applications can achieve excellent performance on a dedicated machine, however, it would be far more cost-effective if these tasks could coexist with other applications in a multiprogrammed environment. Unfortunately, out-of-core tasks have the potential to severely degrade the performance of other tasks which are attempting to use the machine at the same time. This problem arises because operating on massive data sets consumes physical resources (memory and disk bandwidth) at a rapid rate, displacing the working sets of other applications and increasing their page fault service times. To make matters worse, successful prefetching causes physical resources to be consumed even faster, increasing the negative impact on other applications.

In many cases, the excessive resource consumption by out-of-core tasks is caused not by inherent resource requirements, but rather by sub-optimal resource management policies in the operating system. While

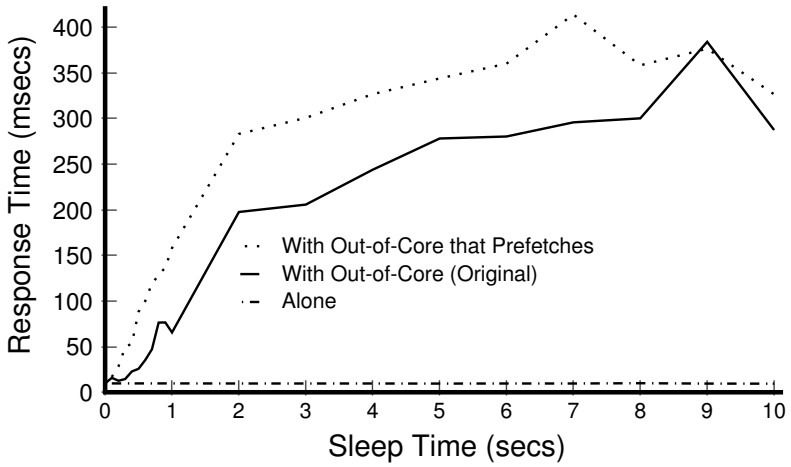


Figure 4.1. Impact of sharing the machine with an out-of-core matrix-vector multiplication (MATVEC) on the response time of an interactive task across a range of sleep times between touching 1 MB of data.

the default policies perform well in most cases, they are poorly suited to the demands of memory-intensive programs. For instance, most commercial operating systems use a global page replacement algorithm, which allows pages to be stolen from any application to satisfy page faults. Various approximations of a least-recently-used (LRU) policy, or the well-known clock algorithm [15] are common. Interactive tasks are particularly vulnerable in such an environment since they are unable to defend their memory effectively. Consider an editor program which may have no memory system activity for several seconds while it waits for user input. A program computing the inner product of two out-of-core vectors could easily sweep through all of physical memory in this time, stealing pages from the editor as they move to the head of the LRU queue. In this case, the out-of-core computation could have achieved the same performance using only two pages of physical memory, allowing the editor to retain its pages and remain responsive regardless of the intervening delay.

To illustrate the impact of out-of-core applications on interactive performance, we ran the following experiment on a 4-processor SGI Origin 200 configured to have approximately 75 MB of memory available to user programs. A simple program emulates the memory system behavior of an interactive task by repeatedly touching a 1 MB data set, then sleeping for a fixed amount of time. By varying the amount of sleep time we can control the frequency with which each page of the "interactive" task is accessed. The "response time" is

the time to touch the entire data set. This program is run concurrently with one that repeatedly performs a matrix-vector multiplication on an out-of-core data set (400 MB). The results are shown in Figure 4.1. With no sleep time, the "interactive" task defends its memory extremely well, achieving the same response time as on a dedicated machine. As the sleep time increases, however, the task incurs an increasing number of page faults and the response time rises. When the out-of-core program uses prefetching, the response time of the interactive task begins to increase at much shorter sleep times, grows much faster, and rises to a higher level. Prefetching combined with global replacement puts the interactive task at a serious disadvantage.

In recognition of the shortcomings of existing operating system policies, a significant amount of recent research has focused on customizable operating systems (as discussed in Section 1.2.5). While a customizable operating system could provide the flexibility to tailor the resource management policies for out-of-core codes, our results in this thesis demonstrate that we can achieve the desired outcome (i.e. customizable behavior) in this particular case through relatively modest extensions of today's commercial operating systems. The role of the operating system continues to be the global allocation of resources across all applications, while the role of each out-of-core application (via the compiler and run-time layer) is to effectively manage the resources it has been granted.

4.2 Alternatives for Memory Management

The goal of a virtual memory management system in a multiprogrammed environment is to share the physical memory resources among all the competing applications. Most operating systems provide policies that perform well in the common case, but exhibit bad behavior when a memory-intensive program is sharing the machine with others. In this section we discuss why it may be beneficial to give demanding applications control over their own memory management, and examine some forms such control could take.

4.2.1 Global vs. Local Replacement

An out-of-core task can degrade the responsiveness of an interactive task because global replacement policies select victims from among all the pages in the system without regard to ownership. In contrast, a local page replacement strategy helps to isolate each process from the paging activity of others. Each

process is allocated a fixed set of physical pages and a victim is selected from among them as needed. Thus, interactive tasks would not have to worry about losing pages to a demanding out-of-core program. Unfortunately, poor memory utilization may occur, as pages are not allocated to processes according to their need. Attempting to determine the right number of pages to allocate to each process and dynamically adjusting this number during execution can improve memory usage but greatly complicates the operating system. A second, and more serious problem, with dynamically adjusting the allocations is that out-of-core programs may still be allocated too many pages at the expense of interactive tasks. For instance, re-allocation strategies based on page fault frequency [14] could be fooled by the consistently-high page fault rate of an out-of-core application. In practice, most workstation operating systems use global page replacement.

Although local replacement policies can help to insulate processes from each other, they may not provide the best replacement policy for each application. Rather than altering the overall strategy employed by the operating system, it is preferable to modify individual applications so that their competition for physical resources better reflects their actual needs. This approach enables applications to improve their own performance through local replacement decisions that are superior to those used by the operating system. The largest drawback of specializing applications to do memory management is the burden placed on the programmer; however, in our framework all the necessary modifications are performed automatically by the compiler.

4.2.2 Application-Managed Replacement

Giving specialized applications more control over their own memory management to improve their performance has been suggested before. For instance, the Mach operating system supports external pagers to allow applications to control the backing storage of their memory objects [44]. Extensions to the external pager interface have been used to implement user-level page replacement polices [38], and to support discardable pages (i.e. dirty pages that do not need to be written to backing store) [51]. In contrast, our approach shows that specialized applications can and should exploit extra control for the benefit of other applications executing concurrently. This is especially true for programs that use prefetching to improve their own perfor-

mance since the gains they enjoy impose a heavy penalty on other processes sharing the system. In this case, the operating system could require that prefetching applications also explicitly release pages.

Given that application-controlled memory management is desirable, one possibility is for the operating system to allow applications to choose from a small set of "reasonable" replacement policies. This strategy does not require much effort on the part of the application programmer, but also does not provide a great deal of power or flexibility. Another possibility is for the operating system to provide a more general interface that allows applications to explicitly specify which of their pages can be reclaimed. This approach is preferable since individual applications can implement a variety of replacement policies tailored to their specific needs.

Application management of memory resources through an interface that allows individual pages to be specified can be either *reactive* or *pro-active*. In a *reactive* approach, the operating system notifies the application when one or more of its pages is about to be reclaimed. The application can then implement its own replacement policy by telling the system which pages to take. This is essentially the approach taken by the VINO page eviction extension [47], for example. A reactive system benefits applications that can make better replacement decisions than the default operating system policy, and has the advantage of delaying the decision until memory actually needs to be reclaimed. Unfortunately, it will not help isolate other applications from a memory-intensive one—the operating system still decides which processes should give up pages.

In a *pro-active* system, an application returns pages to the system *before* they are strictly required, either as soon as they are no longer needed or based on some other criteria such as the amount of free memory. A pro-active approach can obviate the need for the operating system to steal pages by increasing the global pool of free memory, thus providing benefit to all applications sharing the system. Of course, the pro-active approach is not without potential cost to the application using it. If the decision to release memory is made without full knowledge of future accesses, as is typically the case, then the application may give up pages that are still useful.

Our system allows applications to pro-actively return memory to the system on a page-by-page basis, for the mutual benefit of themselves and other concurrently executing applications, without placing any additional burden on the programmer. We now describe the extensions to each of the three parts of our system

(the compiler, the run-time layer and the operating system) that are needed to support the effective use of the *release* operation.

4.3 Compiler Support

To determine whether a given page should be released at a particular point, the compiler attempts to answer the following questions. First, will the page be referenced again in the future (i.e., does the reference have *reuse*)? If not, then a release hint is inserted. Second, is the number of other unique pages that will be accessed before the page is reused less than the expected amount of available memory (i.e., is there also *locality*)? If not, then the page is unlikely to remain in memory, and a release hint is inserted. Otherwise, release hints are not inserted.

4.3.1 Complications with Generating *Release* Requests

There is a certain duality between the analysis for inserting prefetches and releases. In both cases, the compiler attempts to model when pages are being reused, and whether enough intervening accesses exist between these reuses to cause displacement. For prefetching, the question is whether a given page *has* remained in memory since its *last* use (if so, we do not need to insert a prefetch hint for it); for releasing, the question is whether a given page *will* remain in memory until its *next* reuse (in which case we do not want to release it). One difference, however, is that prefetching uses this analysis only to minimize overheads—the latency-hiding benefit of prefetching depends only on scheduling prefetches early enough—whereas the benefit of release hints depends directly on the quality of this reuse analysis.

Ideally, the compiler would be able to analyze the data accesses perfectly and insert these paging directives precisely where they are needed. However, this ideal is not realistic for the following two reasons. First, one cannot always predict memory access patterns with only static information. They may depend on run-time parameters (such as the problem size for the current run) or be data-dependent (such as the indirect references that often occur in sparse-matrix programs, e.g., `a[b[i]]`). While it is possible to issue prefetches for indirect references [21, 40], it is not possible to reason statically about any reuse that they may have, and hence it is not clear that the compiler can generate useful release hints for them. The second major limitation of the

(a) Source code for averaging nearest-neighbors

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    a[i][j] = (a[i+1][j-1] + a[i+1][j]
      + a[i+1][j+1] + a[i][j-1] + a[i][j]
      + a[i][j+1] + a[i-1][j-1] + a[i-1][j]
      + a[i-1][j+1])/9.0;
```

(b) View of data references to the matrix `a`

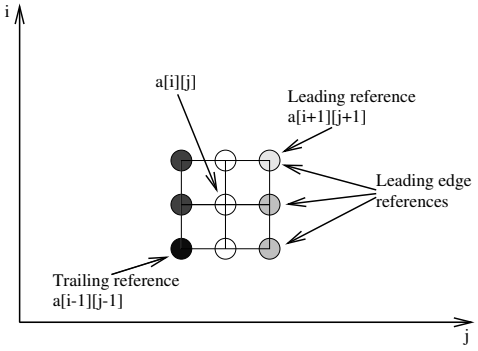


Figure 4.2. Example source code showing multiple references with different types of reuse, and graphical view of the data accesses during a single iteration of the innermost loop.

compiler is that it decides when reuse will result in locality based on an assumption of how much memory will be available to the application at run-time. In a multiprogrammed environment, such assumptions may be wildly inaccurate, especially since the amount of available memory may fluctuate dynamically during execution.

For these reasons, it may be undesirable to actually release a page at the point where the compiler has inserted the corresponding release hint. Instead, the run-time layer should collect information about pages that could be released, according to the compiler-generated addresses, and actually perform the releases only when necessary. In addition to the addresses of releasable pages, the compiler should include some indication of whether it believes the released pages will be used again or not.

4.3.2 An Example of Data Reuse and the Effect on Releases

To help illustrate these concepts, we now present a simple example. Figure 4.2(a) shows the source code for a calculation that averages an element of a matrix with its neighbors, while Figure 4.2(b) depicts the data elements that are touched during a single iteration of the innermost loop. The references have temporal reuse along the `i` dimension (since the items accessed at `a[i+1][*]` are touched again in the next iterations of the `i`-loop). There is spatial reuse along the `j` dimension, and there may also be spatial reuse along the `i`

dimension, depending on the length of the rows.

We can identify two major working sets in this access pattern. At the smallest level, we need to hold the leading edge of the data access square (those references indexed by `j+1`) in memory, requiring at most one page for each of the three references on this edge. Except at page boundaries, the references indexed by `j-1` will fall on the same page as this leading edge due to spatial reuse. We therefore need at most six pages to fully exploit the spatial reuse along the `j` dimension. The second level working set exploits the temporal reuse along the `i` dimension, requiring us to hold three rows of the matrix in memory, so that the row first indexed by `i+1` in one iteration will still be available for the `i` and `i-1` references in the subsequent iterations. Of course, there is also a third level, which corresponds to keeping the entire matrix in memory.

The compiler can determine precisely which references to prefetch and release if it has the dimensions of the matrix and a good estimate of the physical memory available. To successfully exploit the reuse across iterations of the `i` loop, we need to retain three rows of the matrix in memory. If this is possible, then a prefetch will be inserted only for the leading reference, `a[i+1][j+1]`, and a release will be inserted for the trailing reference, `a[i-1][j-1]`. This corresponds to keeping the second level working set in memory. If the amount of memory needed to hold three rows is less than the amount available, the compiler will instead decide to prefetch all three references on the leading edge of the data access square (i.e. the `a[i+1][*]` references) and release the references on the trailing edge, corresponding to the first level working set. If the dimensions of the matrix are unknown at compile-time, the compiler must choose between these two options. Since over-estimating the ability of memory to retain data leads to missed opportunities (both for prefetching and releasing), it is preferable to assume that only the smallest working set will fit in memory. The run-time layer is responsible for reducing the overhead of unnecessary operations that result.

4.3.3 Implementation of Compiler Analysis

Our previous discussion of the compiler implementation in Section 3.1 described how reuse and locality analysis is used to identify references that should be prefetched and released, and how these operations are scheduled using loop splitting and software pipelining techniques. Here, we describe how that implemen-

tation was extended to also encode reuse information into the release hints, allowing the run-time layer to choose which pages to release first. Note that for indirect references (e.g., `a[b[i]]`), we do not insert a release request since it is too hard to predict whether the data will be accessed again.

In addition to identifying the addresses of data that can be released, the compiler also indicates whether the data has temporal reuse, and how soon the reuse is expected, based on the reuse analysis. (Recall that releases may be generated because the reuse is not expected to result in locality). The reuse information is encoded as a priority value which is passed as a parameter in the release requests; larger numbers represent references with earlier reuse—i.e. those which we would most prefer to retain in memory. The release priority is calculated as follows. Let *depth(*i*)* denote the depth of loop i, with the outermost loop nest having a depth of 0. Let *temporal(*x*)* be the set of nested loops in which reference x has temporal reuse. The release priority is computed by the following equation:

$$\textit{priority}(x) \quad = \sum_{i \in \textit{temporal}(x)} 2^{\textit{depth}(i)} \tag{4.1}$$

The run-time layer can use this information to prioritize which pages are actually returned to the system when the memory usage approaches the upper limit, attempting to retain those pages that will be reused earlier to reduce the total amount of paging.

Figure 4.3 shows an example of the output of our compiler for a set of loops that repeatedly perform a matrix-vector multiplication. The compiler analysis has determined that references to the `b` array have temporal reuse with respect to both the `i`-loop and the `iter`-loop, but that this reuse is not expected to result in locality since the volume of data accessed between reuses is more than the memory size parameter. In contrast, references to the `a` array have temporal locality with respect to the `iter`-loop only. Both array references have spatial reuse (and locality) causing the compiler to schedule prefetches for the first reference to each page, and releases after the last reference to each page. Using equation (4.1), a release priority of 1 is assigned to the releases for the `a` array, and a priority of 3 is assigned to the releases for the `b` array, indicating that `b`'s pages will be reused before `a`'s pages. Neither prefetches nor releases are inserted for the `c` array

(a) Original Code

```
int a[100][1000000];
int b[1000000];
int c[100];

for (iter = 0; iter < 10; iter++)
  for (i = 0; i < 100; i++)
    for (j = 0; j < 1000000; j++)
      c[i] = c[i] + a[i][j]*b[j];
```

(b) Code with Prefetch and Release

```
for (iter = 0; iter < 10; iter++) {
  for (i = 0; i < 100; i++) {
    prefetch_block(&a[i][0], 56, 1, 0);
    prefetch_block(&b[0], 56, 3, 3);
    for (j1 = 0; j1 < 770048; j1 += 16384) {
      prefetch_release_block(&a[i][245759 + j1],
                &a[i][j1-16384], 4, 1, 2);
      prefetch_release_block(&b[245759 + j1],
                &b[j1-16384], 4, 3, 5);
      for (j = j1; j < j1 + 16384; j++)
        c[i] = c[i] + a[i][j]*b[j];
    }
    for (j = 770048; j < 1000000; j++)
      c[i] = c[i] + a[i][j]*b[j];
    release_block(&a[i][770048], 56, 1, 1);
    release_block(&b[770048], 56, 3, 4);
  }
}
```

Figure 4.3. Example of the output of the prefetching compiler. Arguments are: (prefetch address, release address, number of 16KB pages, release priority, request identifier)

since this item is smaller than a page and is expected to remain in memory.

4.4 Implementation of Operating System Support for Release

As described in Section 3.3.1, the operating system handles the prefetch and release requests, and maintains the shared page which is used primarily as a bitmap, indexed by virtual page number, in which bits are turned on to indicate that the corresponding page is in memory, and cleared otherwise. To allow the run-time layer to make intelligent choices about when to release memory, the operating system also uses the shared

page to indicate the current number of pages in use by the process, and the upper limit on pages that the process should be using. The first two words in the shared page are reserved for this purpose.

The estimates of current and maximum usage are updated only when the process experiences some type of memory system activity, rather than every time the information changes. One consequence of this approach is that an application's upper limit may drop dramatically if another process begins using memory (reducing the total free memory in the system), but the first process will not be informed of this change until it issues a prefetch/release request, page faults, or has memory stolen from it. The alternative approach of immediate updates would require the operating system to either maintain a list of processes that should be informed, or to scan the list of all processes each time the amount of free memory in the system changes. This additional expense does not appear to be justified.

4.4.1 Setting the Memory Limit

The goal in setting the upper limit on memory usage is to prevent the default page replacement policies from being activated, if at all possible. IRIX provides a number of tunable system parameters that control when pages will be stolen; these parameters can be also used by the *PagingDirected* policy module in an effort to prevent such activity. First, the maximum number of pages that any process can have resident in memory (*max_rss*) can be set. If a process exceeds this limit, the system paging daemon will attempt to trim physical pages from it. Second, the minimum number of pages that should be kept free (*min_freemem*) can be set. If total free memory falls below this limit, the paging daemon will steal pages from all processes in the system according to an approximation of an LRU policy.

If physical memory is ample, it is sufficient to tell the process to remain below *max_rss*. When memory is limited, the process should be encouraged to use no more than its current memory usage (*current_size*), plus the amount of free memory in the system (*tot_freemem*), less *min_freemem*. The recommended upper limit on memory usage in our system is thus given as follows:

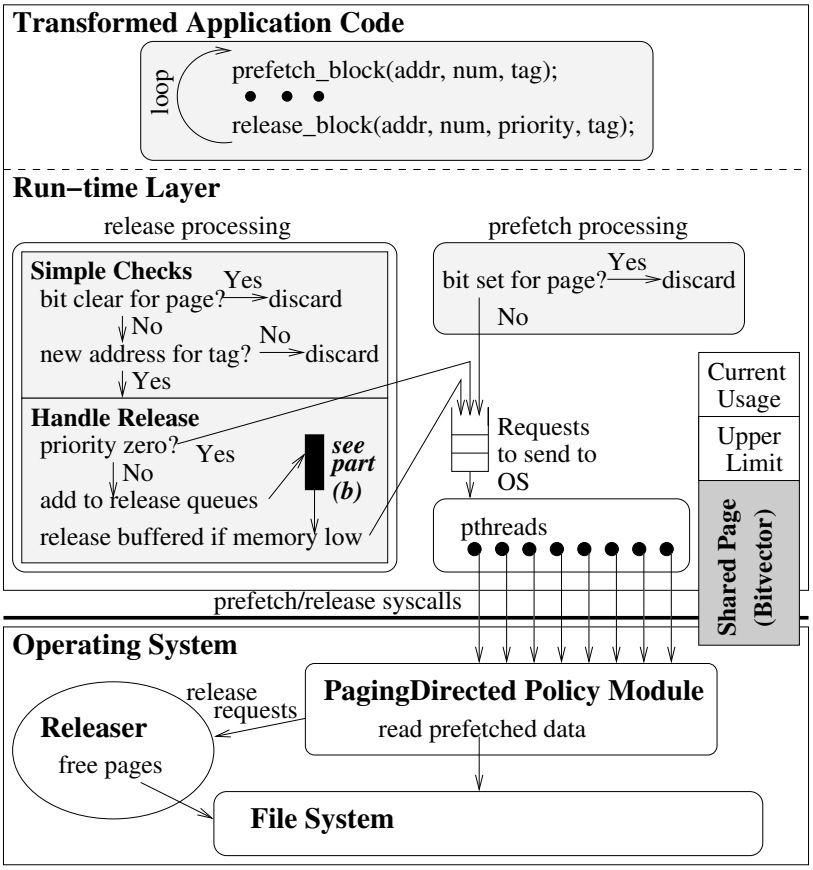
$$\textit{upper limit} \quad = \quad \min(\textit{max\_rss}, (\textit{current\_size} + \textit{tot\_freemem} - \textit{min\_freemem})) \tag{4.2}$$

Note that in setting this upper limit we are not guaranteeing that the application will be able to allocate this many pages for itself. Instead, the upper limit is an indication of the number of pages for which the application is allowed to compete. Pages that have already been allocated to another process are not part of the global free memory pool and thus may not be acquired by the prefetching application. One result of this decision is that the upper memory limit is a moving target which is dynamically adjusted as the total demand for physical memory by all applications changes. Thus, the OS does not try to determine the "right" amount of memory to allocate to each process, it simply tells interested processes how much memory is still available.

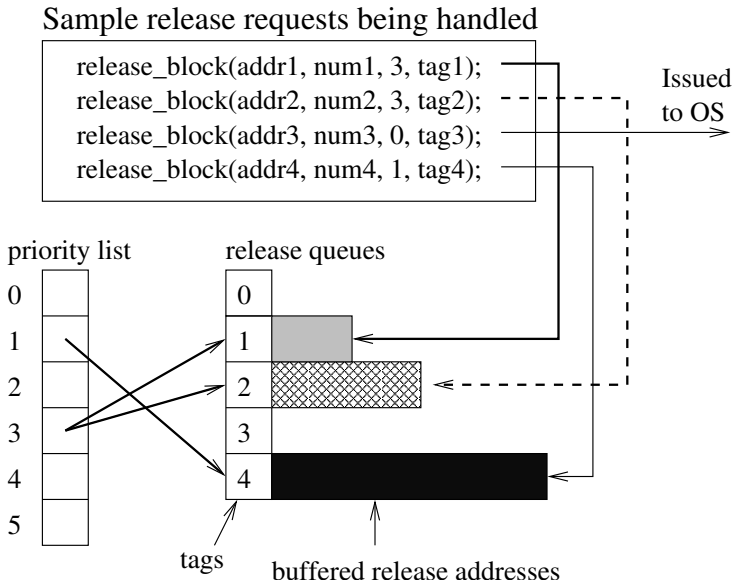
4.5 The Run-Time Layer Support

The role of the run-time layer is to use the information provided by the operating system and the compiler to answer the following questions: When should memory be returned to the operating system? How many pages should be released? Which of the "releasable" pages should actually be given up?

The decision of when to release memory depends primarily on how close the application is to the upper limit on memory usage suggested by the operating system. The decision of how much memory to release is more complicated. The run-time layer needs to balance the desire to remain below the operating system limit, the desire to retain as much memory as possible, and the desire to perform release operations as infrequently as possible to minimize overhead. For example, suppose the run-time layer detects that the application is close to its upper memory limit, and has knowledge of 1000 pages that could be released. By releasing all of these pages, the run-time layer increases the amount of time before it will have to act again, but it may have given up pages that would be used again in the future by acting too aggressively. The run-time layer should also consider the application's expected future need for memory when deciding how much to release. If the application is close to the upper memory limit, but only needs a small number of additional pages, the run-time layer may not need to release memory at all. Finally, once the run-time layer has determined that a release is necessary, and has decided how many pages to release, it must choose which pages should actually be returned to the operating system. This decision depends on the expected future use of these pages; the run-time layer's choice should be guided by information from the compiler.



(a) Processing of prefetch and release requests in the run-time layer.



(b) Buffering of release requests using tags and priorities assigned by the compiler.

Figure 4.4. Handling prefetches and releases at run-time.

There are two situations that may arise from the compiler analysis. First, the compiler may have inserted release hints because it has determined that the page will not be reused again. The run-time layer should release these pages before any pages that are known to have reuse. Second, the compiler may have detected that data reuse existed, but inserted release hints anyway because the volume of data accessed between reuses was expected to flush the page from memory. For these pages, the run-time layer should perform releases according to the intrinsic data reuse (which can be revealed by the compiler), attempting to keep as much data in memory as possible for the subsequent accesses. For instance, suppose the application is repeatedly accessing an array that is much larger than physical memory. The run-time layer can implement *most recently used* (MRU) replacement once the memory usage approaches the upper limit set by the operating system, thus keeping at least the first portion of the array in memory for future use.

4.5.1 Implementation Details

Figure 4.4 illustrates how prefetches and releases are processed by the run-time layer. Part (a) of the figure is the same as Figure 3.20, except that the release processing component has been expanded. In Figure 4.4(b), we illustrate the use of priority values and queues to buffer releasable pages in the run-time layer.

The same set of pthreads which handle prefetch requests are also used to actually issue the release requests to the operating system. We have built run-time layers which implement two different policies for handling the release requests inserted by the compiler—one aggressively issues release requests to the operating system at the time when they are encountered—this is the policy that was used for the results in Chapter 3. The second policy buffers releases based on the compiler-inserted priorities and only issues requests when necessary, based on the information provided by the operating system. By comparing these two approaches, we can evaluate the usefulness of buffering release requests in the run-time layer rather than simply relying on the compiler analysis.

In both cases, the run-time layer attempts to reduce overhead by filtering out the obviously bad releases inserted by the compiler. There are two ways in which these bad releases are detected. First, the requests inserted by the compiler are checked against the bitvector to make sure that the pages are in memory. Second,

the run-time layer tracks the last address released for each unique release directive placed in the code, using the request identifier (or tag) generated by the compiler. The first release request for any tag is recorded until the next request for that tag is issued. If a release request identifies the same page as the previous request, it is dropped since the page is obviously still in use. If instead, the current release request identifies a different page, then the previously recorded release is actually handled and the current one is recorded. The releases issued by the run-time layer are thus always one or more iterations behind those identified by the compiler. Handling a previously recorded request involves either placing it in a release queue (if buffering is being used), or issuing it to the operating system. Programs with loop nests that have unknown bounds often cause the compiler to generate overly-aggressive code, and these simple checks help to reduce the overhead of releasing pages that are still in active use.

Figure 4.4(b) shows how release requests are buffered. Requests with no reuse (i.e. a priority of *0*) are issued to the OS after passing the simple checks. Other requests are stored in release queues indexed by their tags, allowing multiple buffered releases for a particular reference to be coalesced into a single entry in the queue. When the first release for a tag is seen, the priority value is used to index into the priority list where a pointer is set to the release queue for that tag. The priority list can hold pointers to multiple queues having the same priority. When a release request is placed into one of the queues, the current memory usage and memory limit are checked. If the current usage is close to the limit, the priority list is used to issue releases from the lowest-priority queues. Requests are issued from all queues at the same priority level in a round-robin fashion. Currently, the run-time layer attempts to release a total of 100 pages whenever releasing is deemed necessary.

As we will show in Section 4.6, even the simple strategy of always issuing the releases improves the performance of the prefetching out-of-core application over prefetching alone, while simultaneously keeping memory free for other applications in most cases. When there is temporal reuse in an application, however, the advantages of prioritizing releases become clear.

Table 4.1. Description of applications.

Name	Description	Input Data Set	Memory Required (and % of Available)	Orig Exec. Time (mins)
BUK	integer bucket sort algorithm	2^{24} 20-bit integers	206 MB (275%)	13.5
CGM	sparse linear system solver	40k x 40k sparse matrix, ~15M non-zeros	206 MB (275%)	16.2
EMBAR	monte-carlo simulation	2^{24} random numbers	134 MB (179%)	13.8
FFTPDE	3-D FFT PDE	256x128x128 complex matrix	235 MB (313%)	34.2
MGRID	computes 3-D potential using multigrid solver	256x256x256 matrix	452 MB (600%)	23.9
MATVEC	matrix-vector multiply	10^2 x 10^6 matrix, 10^6 vector	404 MB (539%)	11.1

4.6 Experimental Results

To evaluate the concepts presented in this chapter, we ran several out-of-core applications with the simulated interactive task described in Section 4.1. The platform used to obtain these results is our commercial system, IRIX, described in Section 2.4.3. We begin with a look at the impact of prefetching, alone and with both aggressive releasing and release buffering, on the execution time of the out-of-core program. To explain the basic performance results, we will then take a closer look at the effectiveness of the release operation by examining the activity in the virtual memory subsystem. Finally, we evaluate the usefulness of explicitly releasing memory for improving the response time of the interactive task.

4.6.1 Benchmarks

We performed our experiments using out-of-core versions of five applications taken from the NAS Parallel benchmark suite [6] as well as a matrix-vector multiplication kernel (MATVEC). The code for MATVEC was shown earlier in Figure 3.1(a). We have increased the data sets of the NAS benchmarks to make them larger than the available memory on our system. Other than increasing the data set sizes, we did not modify these applications by hand in any way—all prefetch and release operations were inserted automatically by our compiler pass.

Table 4.1 summarizes the characteristics of these applications; each exhibits different data access behavior. EMBAR has only one-dimensional loops, while MATVEC has multi-dimensional loops with known bounds. For both, the compiler analysis is essentially perfect and excellent results are obtained for both the benchmarks themselves and the interactive task. BUK and CGM are more difficult cases, as they involve both unknown loop bounds and indirect references, both of which reduce the compiler's ability to analyze the data accesses. Nonetheless, the run-time layer is able to adapt the behavior based on dynamic conditions and excellent results are again achieved. MGRID and FFTPDE are the most difficult cases. Both involve multi-dimensional loops with unknown bounds. In MGRID the loop bounds change dynamically on different calls to the same procedures, making it impossible to release memory optimally in all cases, since we only generate a single version of the code. In FFTPDE, the access stride changes within a set of loops, making it seem as though the access is not dependent on the loop induction variable. This causes the compiler to identify some releases as having reuse when in fact none exists. Ultimately, the solution to the problems experienced by MGRID and FFTPDE is to generate more adaptive code, and specialize the loops at run-time according to dynamic conditions. Even without this extra sophistication, MGRID performs better with releases and can significantly reduce (although not eliminate) its negative impact on interactive response time. In Chapter 5, we consider how improvements to the compiler scheduling algorithm can generate code that adapts better to dynamic conditions, improving the usefulness of both prefetching and releasing.

4.6.2 Performance of the Out-of-Core Applications

The goal of I/O prefetching is to improve the execution time of out-of-core applications by hiding the page fault latency. The goals of explicitly releasing memory are to reduce the number of page faults in out-of-core programs by making better replacement decisions, to reduce the interference caused by the OS selecting victims for replacement, and to alleviate the impact of out-of-core programs on other applications sharing the same system. We begin by examining how well our scheme achieves these goals from the perspective of the out-of-core applications.

In Figure 4.5, we show the execution times of the out-of-core programs, normalized to the original case.

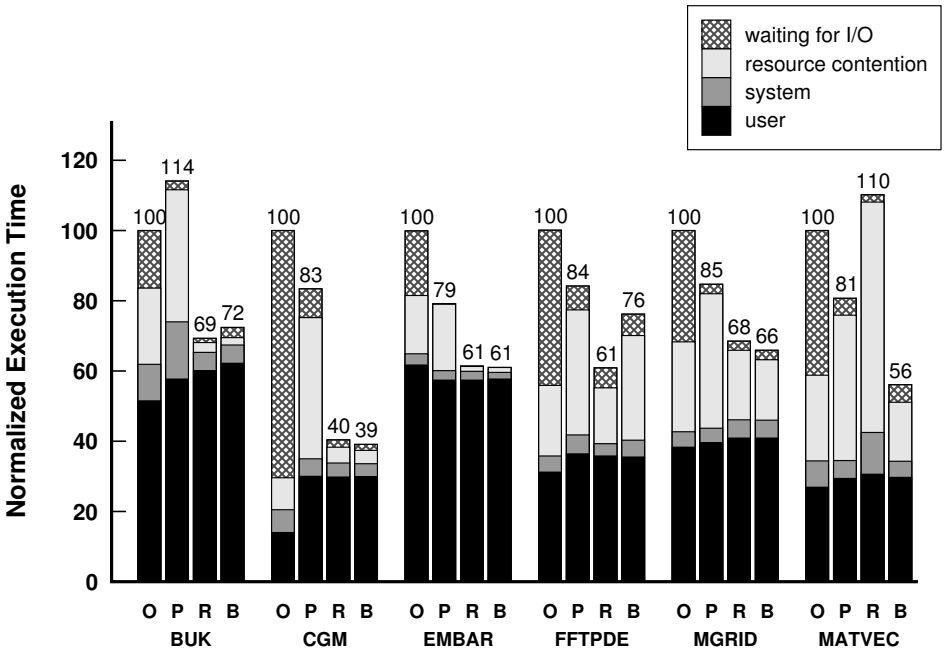


Figure 4.5. Impact of prefetching and releasing on the execution times of the out-of-core applications. (O = original, P = with prefetching, R = with prefetching and releasing, B = with prefetching and release buffering)

For each benchmark we show four bars: the original, unmodified program (**O**), the program compiled to use prefetching only (**P**), the program compiled to use both prefetching and aggressive releasing (**R**), and the program compiled to use both prefetching and release buffering (**B**). Each bar is broken down into four components. The top section is the time that the program was stalled waiting for I/O. The next component is the time that the process was stalled waiting for unavailable resources, including physical memory, memory system locks, and CPUs. The second-lowest component is the system time, which is primarily spent handling page faults. The bottom section of each bar is the time spent executing user code. Increases in user time over the original case show the overhead of handling prefetch and release requests in the run-time layer. Note that the **O** and **R** bars correspond to the experiments shown in Section 3.3.2, although the actual results were obtained on an earlier version of the system software, and should not be compared directly.

All prefetching versions of the benchmarks achieve similar reductions in the I/O stall time, with over 85% of the I/O stall eliminated in all cases. Also, the time spent executing system code is nearly identical across all versions of the benchmarks, and only modest increases in user time occur in the prefetching versions. The

increase in user time is most pronounced for CGM, where a very large number of unnecessary prefetch and release requests need to be filtered out by the run-time layer. These unnecessary requests are the result of the compiler's inability to reason about the amount of data accessed in loops with unknown bounds. For CGM, most of these loops are small and prefetches and releases are not needed. In all cases except for FFTPDE and MATVEC, the results for aggressive releasing and release buffering are very similar, since these applications do not have temporal reuse within a single set of loops, and the compiler analysis is unable to detect reuse across independent sets of loops. When all release requests have zero-priority, both implementations of the run-time layer perform the same actions (issuing the requests to the OS without buffering), although the version which attempts to buffer requests incurs a small amount of additional overhead to check the priorities. In FFTPDE, the compiler incorrectly identifies some references as having temporal reuse, causing the run-time layer to preferentially retain these pages in memory to the detriment of others. For MATVEC, however, the benefit of buffering and prioritizing releases is dramatic. In this case, without buffering, both the matrix and the vector are released, but the vector is frequently reused shortly thereafter. Large amounts of contention occur between the release daemon attempting to free the pages of the vector and the application attempting to reclaim them. When the run-time layer buffers and prioritizes the releases, only the pages of the matrix need to be released and contention is greatly reduced. In the remainder of this section, we will discuss both releasing versions of the benchmarks together, since their behavior is essentially the same, making specific reference to MATVEC in the cases where buffering makes a difference.

Using our experimental HURRICANE platform, we had previously found that releasing memory provided no significant benefit to the out-of-core applications over prefetching alone. Our results here, in contrast, show that there is a substantial reduction in the execution time of the out-of-core applications when releasing is applied aggressively. The speedups from applying both prefetching and releasing over prefetching alone range from 13% for EMBAR to over 50% for CGM. This added benefit is rather unexpected, both because it did not occur in the previous study, and because the run-time layer implementations are not trying to actively improve the replacement policy (since there is no known reuse)—they simply try to maintain as large a pool of free memory as possible by releasing pages which the application apparently no longer needs. There

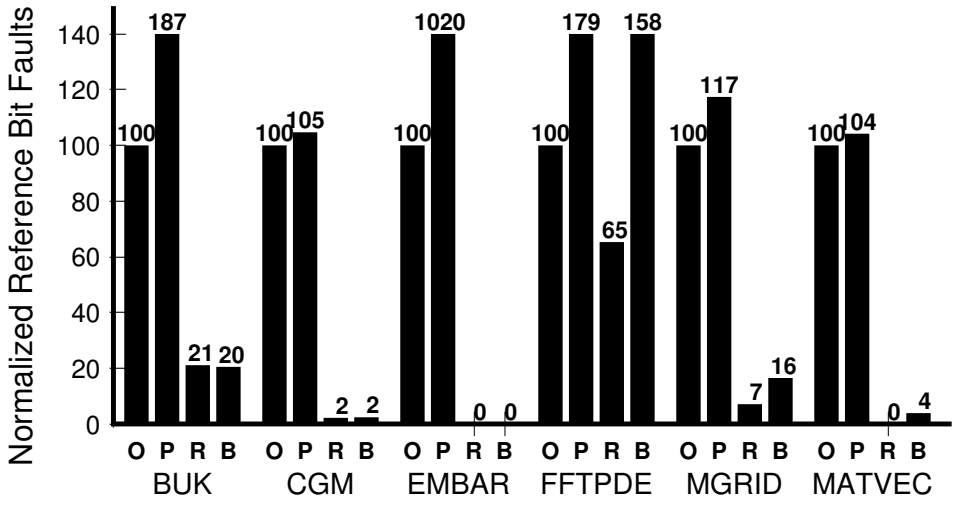


Figure 4.6. Soft page faults due to page invalidations.

are essentially three reasons for the improvement due to aggressive releasing: (i) a reduction in the number of soft page faults caused by the paging daemon attempting to identify unused pages; (ii) a reduction in the contention for memory locks needed by both the fault handling code and the paging daemon; and (iii) improvements in the replacement policy created by the compiler analysis alone. We now discuss the impact of each of these effects.

Looking at the components of the bars in Figure 4.5, we see that the greatest difference between the prefetching-only and the two prefetching-and-releasing cases is in the time stalled for unavailable resources. Without releasing, the paging daemon needs to determine which pages should be reclaimed. To do so, a variant of a clock algorithm is used, in which pages can be reclaimed if they have not been referenced for a number of passes of the clock hand. Since the MIPS TLB does not have reference bits, reference information must be simulated in software using the valid bit instead. As free memory becomes low, pages are periodically marked invalid to see if they are still in use. These invalidations increase the number of soft page faults as the process references, and needs to re-validate, the pages that were still in its working set. However, with aggressive releasing, the paging daemon does not need to find pages to reclaim, thus greatly reducing the number of invalidations.

Figure 4.6 shows the number of page faults caused by these periodic invalidations for each version of

our out-of-core benchmarks. Not only are the number of soft page faults greater when prefetching is used without releasing, the time to service each of these faults is also amplified due to increased contention for locks between the paging daemon and the fault handling code. The time to handle hard page faults is also increased by this contention. When the paging daemon needs to invalidate or reclaim pages, it holds locks on the address spaces of the processes from which pages are being stolen. During this time, page faults for these virtual memory regions cannot be serviced. The releasing daemon must hold the same locks while freeing the explicitly released pages; however, it typically operates on smaller blocks of pages, so the locks can be held for much shorter periods of time. Furthermore, the releasing daemon has been specialized for the purpose of freeing pre-identified pages. Thus, it requires fewer locks overall and can do much less processing per page while locks are held. The resulting lock contention caused by the releasing daemon is significantly less than that caused by the paging daemon.

Finally, in some cases the compiler analysis is able to improve upon the replacement policy without extra support from the run-time layer. In BUK, the data set consists of two very large sequentially-accessed arrays and a third equally large randomly-accessed array. The compiler inserts releases for the first two, but does not try to release the third because it cannot reason about any locality that may exist. The result is that demand for new pages is satisfied by the releases of the first two arrays and the pages of the third array are able to remain mostly in memory. Without releasing, the paging daemon reclaims pages from all three arrays according to their last use, but without regard to their access patterns, causing many more page faults to occur. Although the run-time layer is not able to prioritize releases due to a lack of temporal reuse, the decision by the compiler to not release randomly accessed data effectively accomplishes the desired effect. Having discussed the overall performance impact of our system, we now take a closer look at how effective the compiler and run-time layer are at generating and managing releases.

4.6.3 Effectiveness of Releases

There are two considerations when evaluating the effectiveness of the release operation. First, the purpose of issuing releases is to maintain a large enough pool of free memory to prevent the default page reclamation

Benchmark	Pages Stolen by System	System Page Reclamation Events	Stolen Pages Rescued	Pages Freed by release	Released Pages Rescued	Total Pages Allocated
	Original					
BUK	126,842	2,796	32,532	N/A	N/A	131,354
CGM	289,696	6,130	3,472	N/A	N/A	313,522
EMBAR	126,793	2,987	4	N/A	N/A	165,838
FFTPDE	330,490	7,847	9,999	N/A	N/A	389,504
MGRID	313,595	7,555	806	N/A	N/A	376,301
MATVEC	272,541	11,679	7,159	N/A	N/A	281,297
	With Prefetch and Release					
BUK	5,043	111	4,340	33,916	3,176	158,210
CGM	1,567	34	109	72,276	266	305,805
EMBAR	0	0	0	32,712	4	132,170
FFTPDE	134,612	3,172	16,574	81,520	2,801	395,478
MGRID	72,883	1,735	111	255,114	183,835	360,599
MATVEC	0	0	0	105,588	261,100	286,294

Table 4.2. Pages freed by system or by release, and pages rescued from the free list.

behavior. To see how well we achieve this goal, we look at how much work the paging daemon performs, both with and without releases. Second, we should only be releasing pages that are really no longer in use by the application (or will not be used again for a long time) to avoid increasing the page fault rate. To see how useful the releases are, we look at how many released pages are "rescued" from the free list (i.e. returned to the process that was using it). If we are actually releasing pages that are no longer needed, very few pages should be rescued. The page reclamation and allocation activity is summarized in Table 4.2 for the original out-of-core programs and the versions that both prefetch and release memory without buffering.

From Table 4.2, we see that releases are usually very effective at reducing the need for the paging daemon to reclaim memory. In the worst case, the number of times that the paging daemon needs to operate is reduced by more than half, and the total number of pages stolen is reduced by more than a factor of three. In the other cases, the activity of the paging daemon is reduced by one to two orders of magnitude, both in terms of frequency and number of pages stolen. Although it is very difficult for the application to release its pages perfectly, it can still provide a great deal of assistance to the OS.

Next we look at how often useful pages are reclaimed too early, either by the paging daemon or due to explicit release requests. There are two possibilities. First, useful pages may still be on the free list when they are referenced again, and can be rescued and returned to the application. Second, useful pages may have

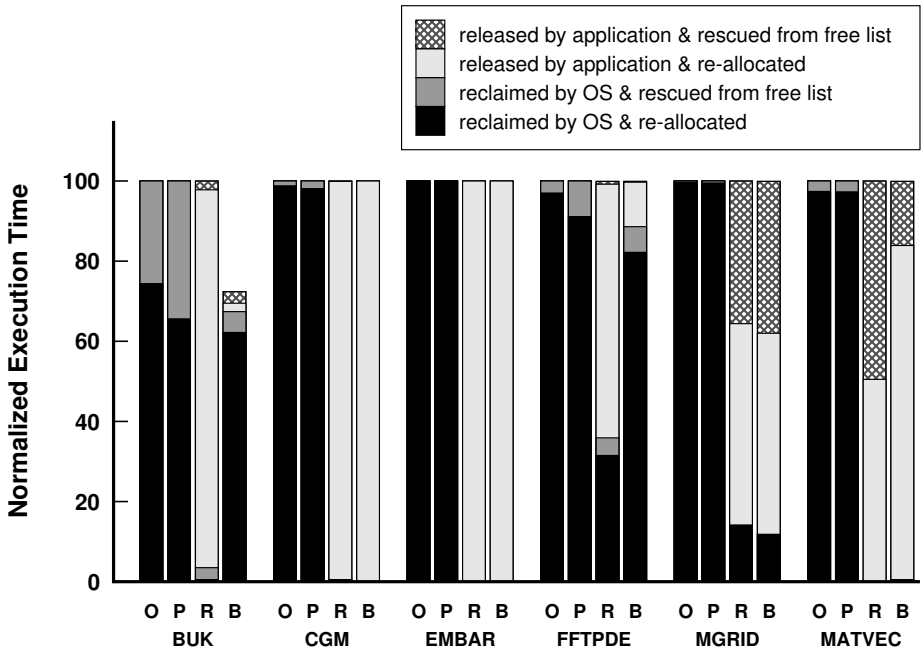


Figure 4.7. Breakdown of outcomes for freed pages.

been re-allocated to hold other data before being referenced again, and the reused data will need to be brought back into memory from swap.

Figure 4.7 shows what fraction of all the pages freed are freed by the paging daemon vs. the fraction freed explicitly by release requests. We also show the fraction of each that are rescued from the free list. The interesting cases here are BUK, MGRID and MATVEC. As we see in Figure 4.7, BUK without any releasing (both the original and prefetching versions) frequently needs to rescue the pages reclaimed by the paging daemon from the free list. The greater demand on memory introduced by prefetching increases the need for the paging daemon to reclaim memory, resulting in useful pages being placed on the free list more often. Consequently, the fraction of reclaimed pages that are rescued also increases. With releasing, however, most of the pages are freed by explicit release requests and very few are rescued from the free list. In this case, releasing helps the application to retain its most-needed pages in memory. For MGRID, we see that even with releasing, over half of the pages freed are reclaimed by the paging daemon, and that more than half of the pages explicitly released are rescued from the free list. This suggests that the compiler is unable to determine which pages to release and when for MGRID. Note also that FFTPDE with release buffering performs very few

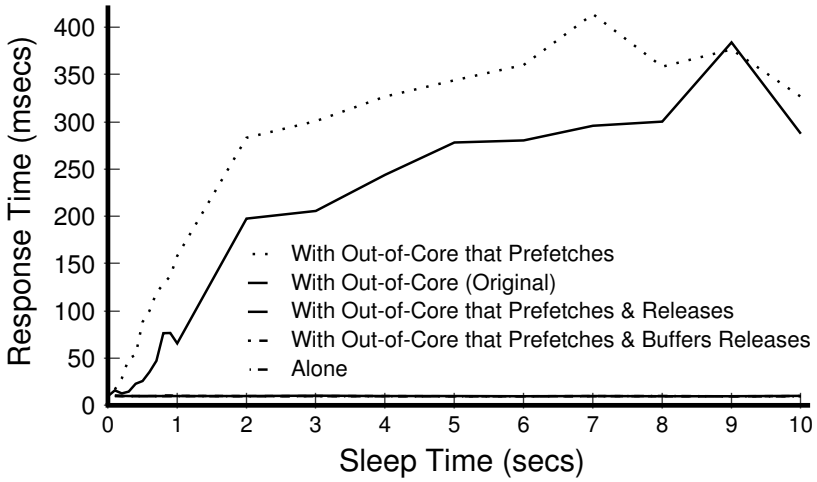
useful releases due to incorrectly attempting to retain pages with no reuse. For MATVEC without releasing, the OS does a reasonable job of freeing the pages of the matrix and keeping the frequently accessed vector in memory. With aggressive releasing, however, approximately half of the pages released are for the vector and need to be rescued from the free list. When release buffering is used, most of the released pages are for the matrix, and the number of rescued pages is much smaller. Overall, we can see that releasing greatly reduces the need for the paging daemon to reclaim memory, and typically does a good job of releasing pages that are no longer in use.

Detecting pages that were freed too early and re-allocated before they could be rescued is a more difficult task. These pages will increase the total number of page allocations required (over the ideal) as new pages are needed to bring the reused data back into memory. While we cannot compare the total number of page allocations to the ideal number, we can look at the number of allocations in the original case versus the prefetching-and-releasing cases. From Table 4.2, we see that the total number of page allocations increases by a small amount with prefetching and releasing in half of the cases, and decreases by a small amount in the other half. This suggests that releasing is typically doing no worse at freeing needed pages than the paging daemon, but results in much less contention.

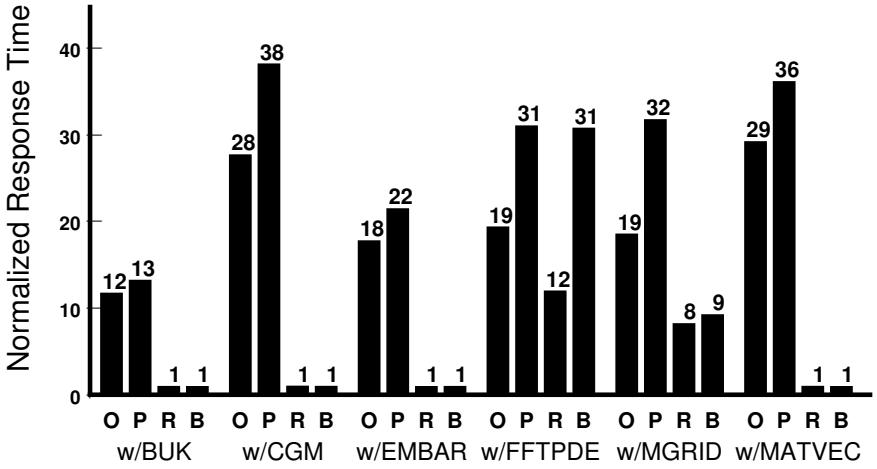
We now look at how useful releases are for improving the performance of the interactive task.

4.6.4 Impact on Interactive Response Time

Figure 4.8 gives an overview of the performance improvements obtained for the "interactive" task. In Figure 4.8(a), we show the average response time for the interactive task when executed concurrently with MATVEC across a range of sleep times. As discussed in Section 4.1, the response times become greatly inflated when the out-of-core program executes normally, and are made even worse when prefetching alone is used. When releasing is added to prefetching, however, the response times of the interactive task almost perfectly matches the times obtained when it is run alone on the machine, regardless of the amount of sleep time. Although blindly following the release directives inserted by the compiler has a severe effect on MATVEC's own performance, this strategy does leave most of memory free for the interactive task. However, when



(a) Impact of MATVEC on response time (last 3 lines in key overlap).



(b) Impact of each out-of-core benchmark on response time at 5 second sleep time, normalized to stand-alone response time of 9.5 msec.

Interactive with Benchmark	Hard Page Faults			
	Original	Prefetch Only	Prefetch & Release	Prefetch & Buffer Release
BUK	25	29	0	0
CGM	61	65	0	0
EMBAR	51	62	0	0
FFTPDE	28	55	28	44
MGRID	30	48	11	11
MATVEC	61	63	0	0

(c) Average number of page faults requiring I/O for the interactive task with each out-of-core benchmark.

Figure 4.8. Impact of releasing on interactive response time.

release buffering is used to improve the performance of MATVEC, there is still nearly no impact on the interactive task. The run-time layer is able to both buffer releases for the benefit of the out-of-core task and keep enough memory free for the interactive one. The negative impact of the out-of-core program on the response time of the interactive task in this case has been almost completely eliminated. For the other out-of-core applications, we chose an intermediate sleep time of five seconds for the interactive task and recorded the average response times. The results for each of the four versions of the out-of-core programs are shown in Figure 4.8(b). The response times in this graph have been normalized to the time for the interactive task executing alone on the machine. As we see in Figure 4.8(b), releasing is usually successful at eliminating or substantially reducing the degradation in interactive response time. FFTPDE with release buffering is the exception as this benchmark fails to release enough memory.

Figure 4.8(c) shows the average number of hard page faults (i.e. those that require I/O) experienced by the interactive task during a single sweep through its data set, when it is executed concurrently with each version of our out-of-core benchmarks. From this table, we see that the number of page faults increases when the out-of-core program uses prefetching alone, rising to the maximum level of 65 pages. At this point, the entire data set of the interactive task must be paged in from the swap space. When the out-of-core program also releases pages, the number of hard page faults is significantly reduced. This result verifies that the primary reason for the increased interactive response time is not being able to keep pages in memory.

4.7 Summary

Our investigation of pro-actively releasing memory, based upon inserting release hints at compile-time and exposing the results of the compiler's reuse analysis to the run-time layer, has produced a number of interesting results.

First, we found it was beneficial to the performance of an out-of-core application to explicitly release memory, even without using buffering to improve the replacement policy. We found a surprisingly high degree of contention between the system page daemon and our applications, as each needed to manipulate page tables, and perform I/O. The IRIX page daemon is designed with the assumption that paging is expensive;

thus when the system runs out of memory and pages need to be reclaimed, performance is expected to be poor. The addition of prefetching to hide the latency of page faults changes this underlying assumption, and the manner in which pages are reclaimed needs to be carefully considered.

Second, we found that there are situations in which delaying the decision to release memory, and re-ordering the pages to be released, can be a significant benefit. By considering both the inherent reuse of data, and the global memory pressure, the run-time layer can begin to implement an application-specific replacement policy that is closer to optimal.

Finally, we found that an out-of-core program does not have to cause problems for interactive applications sharing the machine. The same techniques for explicitly releasing memory that improve the performance for the out-of-core program also leave enough memory free for the interactive task. By limiting the work done by the page daemon, we reduce the chances that pages will be stolen from applications that are still using them.

Despite these encouraging results, we see that generating useful release information suffers from the same problems as generating prefetches—many important quantities are unknown at compile-time and the compiler must make a single, static decision in the face of incomplete information. To address these problems, we consider a novel technique for software pipelining around multiple loop nests in Chapter 5.

Chapter 5

Improving the compiler scheduling algorithm

If everything seems under control, you're just not going fast enough. — Mario Andretti

In Chapters 3 and 4 we described our compiler algorithm for prefetching and releasing and showed it to be a promising approach for handling the memory demands of out-of-core numeric benchmarks when combined with run-time and operating system support. We also showed, however, that there were numerous situations where the code transformations made at compile-time could not be adequately adapted to deal with dynamic run-time conditions. In particular, prefetches were frequently scheduled too late to be fully effective, limiting our ability to hide the I/O latency and reduce overall execution time.

In this chapter, we introduce a new compiler algorithm for scheduling prefetch operations and show that this algorithm allows us to handle a much wider range of dynamic conditions. We begin in Section 5.1 with a discussion of the scheduling challenges that our new algorithm needs to address. We then develop our new algorithm incrementally in Section 5.2. Throughout this section, we evaluate the effect of each improvement to the algorithm using simulations and micro-benchmarks. Section 5.3 evaluates the impact of the final algorithm on the performance of the NAS Parallel benchmarks.

5.1 Scheduling challenges

As discussed in Section 3.1, our compiler algorithm for inserting prefetch requests into application source code uses *software pipelining* to ensure that the request for data is issued early enough to hide the latency of a disk fetch. If we can construct an effective pipeline of prefetch requests and data accesses across a set of loops, we can substantially improve performance by avoiding stalls due to page faults.

Software pipelining is a technique commonly applied to expose instruction-level parallelism in loops. Our usage of the technique was adapted from its application to the problem of scheduling prefetches for cache misses in looping codes [41]. In both cases the latencies that need to be hidden by the pipeline are known at compile-time and are small relative to the size of the loop being pipelined. Under these conditions, it is reasonable to compile for a fixed latency when constructing the pipeline and to consider only the innermost loop nest. When software pipelining is used to schedule prefetches for page faults in out-of-core looping codes, however, one or both of these properties may not hold.

Our initial compiler algorithm for I/O prefetching used a fixed latency estimate and attempted to address the problem of small inner loops by introducing the simple heuristic described in Section 3.1.2: construct the pipeline across the innermost loop that accesses a sufficient amount of data [40]. This technique can help to select a loop that is large enough for a given latency, but only if the loop bounds are known during compilation so that the amount of data accessed can be calculated. When loop bounds are symbolic, choosing the right loop for pipelining remains a problem. The combination of nested loops and large latencies creates a third problem for software pipelining of I/O prefetch requests: the pipeline is repeatedly filled and drained on each iteration of the surrounding loop, and the time to initialize the pipeline may be large compared to the time spent in the steady state. We now elaborate on the problems of finding the right prefetch distance and dealing with nested loops.

5.1.1 Variations in Prefetch Distance

The first step in the software pipelining algorithm (for both the original I/O prefetching version, and our new variations) is to determine the *prefetch distance*. This distance is expressed as a number of loop

iterations, and is calculated using an estimate of the I/O latency, and the shortest path through the loop body (see Equation 3.1 on page 52). If our calculated prefetch distance is too large, then we will increase the memory pressure since more pages are needed to hold prefetched data. As a result, evictions may occur earlier than necessary. Over-estimating the prefetch distance by a small amount is not a serious problem, however, because main memory has plenty of capacity to buffer the prefetch requests and we can use the release hints to make good replacement decisions. If, on the other hand, we underestimate the prefetch distance, then we will be unable to hide all of the I/O latency. There is no way for the run-time layer to compensate for a late prefetch request by issuing it earlier—by the time the run-time layer sees the request, it is already too late. There are two basic reasons that we may be unable to estimate the prefetch distance accurately at compile-time: the estimate of the time to execute the loop body may be inaccurate, and/or the estimate of the I/O latency may be inaccurate.

Causes of Inaccurate Estimates

We estimate the time to execute one iteration of the loop body by counting SUIF instructions in the shortest path through the loop. At this point in the compilation process, the instructions are in an intermediate format, so we do not know the actual machine instructions that will be executed at run-time. Some of these instructions may be removed completely by later optimization passes. We make an additional simplifying assumption that each SUIF instruction will execute in one cycle. These inaccuracies could cause the loop to execute faster than expected, resulting in late prefetches which cannot hide all the I/O latency. Even if we knew the actual cycle count for the shortest path, we would still have a problem with calculating the prefetch distance. The actual path through the loop taken at run-time may be many times longer than the worst-case shortest path; this would result in prefetches being issued earlier than necessary.

The I/O latency estimate is a compile-time parameter which we have chosen based on measurements of our target systems. It is expressed in terms of processor cycles, so both the disk speed and the clock speed of the target system are implicitly included in the estimate. Even on a single, uncontended system, however, the worst-case time to read a page from disk could be about twice as long as the average case depending on

the disk seek time and rotational latency required to locate the data. Using an average measured value means that some pages may be requested too early, while others will be requested too late. We could conservatively double the average page fetch time, relying again on the large buffering capacity of main memory, and the release hints to cope with the inflated prefetch distance, but there is another more serious problem. We do not, in general, want to recompile the applications for each target system. Consider, for example, general-purpose computers based on the Intel Pentium IV architecture. Currently, the same binary program could execute on processors that differ by more than 1 GHz in clock speed. At the same time, a particular system built around these processors could include anything from 5400-7200 IDE disks to 10K-15K SCSI disks. It is highly-undesirable (if not completely unreasonable) to have to compile a unique binary for each configuration on which the application will execute, just so the target latency parameter can be specified.

Solution: Variable Prefetch Distances

Rather than compile for a fixed latency estimate and a statically estimated dynamic loop execution time, we would prefer to express the prefetch distance as a variable at compile-time and generate code to calculate it dynamically based on actual run-time behavior. There are numerous options for obtaining a dynamic latency estimate during execution. For instance:

- Obtain a latency estimate by measuring the actual time for a page fetch (this could be done once for a given system and stored in a well-known location such as the /proc filesystem where all interested processes could read it). The actual time to execute the loop body can be obtained using profiling information (such as cycle counters)—the compiler's static shortest-path estimate can be used as an initial value. With these two parameters obtained at run-time, we can now calculate the prefetch distance as the latency divided by the loop execution time as before.

- Track the percentage of late prefetches in a loop and dynamically increase the prefetch distance until the late fraction becomes acceptably small. It would likely be useful in this case to include additional information, such as disk queue length, to avoid continually increasing the prefetch distance in a system that is bandwidth-limited.

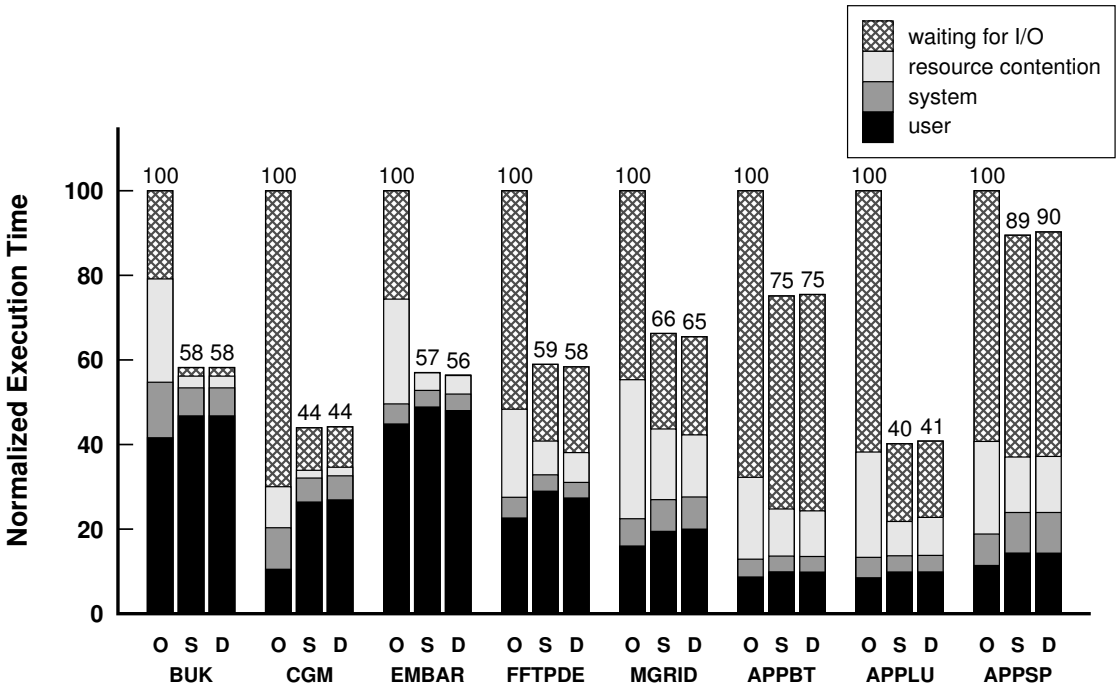


Figure 5.1. Effect of calculating prefetch distances at run-time. Bars labeled "O" are the original, non prefetching version; bars labeled "S" use a static compile-time latency; bars labeled "D" use a dynamic latency value (equal to the static value) obtained at run-time.

Other options are also available, but an important question is whether or not using a variable latency and adding code to calculate the prefetch distances at run-time, rather than calculating these values statically at compile-time, introduces substantial run-time overhead.

To address this question, we modified our compiler to generate code that calculates the prefetch distances dynamically at run-time. The latency estimate is simply entered from an input file, or the command line, since we are primarily interested in evaluating the overhead of performing these calculations at run-time, not in exploring how to measure latency in a running system. We also added code to calculate the length of a loop body at run-time. This calculation uses the compiler's count of the number of SUIF instructions in the loop body and continues to assume that each SUIF instruction can execute in a single cycle, however, we are able to use the run-time values of symbolic inner loop bounds rather than assuming a worst-case execution of a single iteration. Using this version of the compiler, we generated code for the NAS Parallel benchmarks and executed them on our Irix prototype.

Benchmark	Static	Dynamic	% Increase
BUK	21079.6	21082.1	0.01 %
CGM	15811.7	15876.4	0.41 %
EMBAR	50693.4	50719.7	0.05 %
FFTPDE	28245.9	28298.7	0.19 %
MGRID	31563.1	31581.7	0.06 %
APPBT	21731.4	21726.2	-0.02 %
APPLU	11098.4	11104.2	0.05 %
APPSP	44741.5	44795.8	0.12 %

Table 5.1. Graduated instructions for static and dynamic prefetch distance calculations (millions of instructions)

The results of this experiment are shown in Figure 5.1. The original, non-prefetching version (labeled "O") is shown for reference. Bars labeled "S" are the results for prefetching with the statically-calculated prefetch distance. Bars labeled "D" show the results for prefetching with dynamically-calculated prefetch distances (we used the same value for the latency estimate in the dynamic cases as in the static ones).

In most cases, the difference between using a static latency value to compute the prefetch distance at compile time, and using a dynamic value obtained at run-time is negligible. To further quantify the overhead of calculating prefetch distances dynamically, we collected graduated instruction counts for the static and dynamic versions of each benchmark, using the hardware counters provided by the MIPS R10K microprocessor. For these experiments, we removed the actual prefetch and release operations to focus specifically on the overhead of the prefetch distance calculations. The results are shown in Table 5.1. In all cases, the increase in instructions is less than 0.5% confirming that the cost of the dynamic strategy is negligible.

Overall, the potential benefit of calculating prefetch distances at run-time, and the increased flexibility (due to not needing a recompilation if the expected latency changes) make this an extremely useful feature to incorporate into our compiler, independent of any changes to the actual scheduling algorithms. We now consider several alternatives for further improving the performance of prefetching on benchmarks with multi-dimensional nested loops such as FFTPDE, MGRID, APPBT, APPLU and APPSP. We do not consider BUK or EMBAR in the remainder of this chapter, as they do not have multi-dimensional loop nests. We also exclude CGM from further consideration since it has only a single multi-dimensional loop, in which the inner loop bounds depend on array accesses in the outer loop. Handling this case correctly would add significant

complexity to our new scheduling algorithm, and is unlikely to further improve the performance of CGM.

5.1.2 Problems with nested loops

Even under the ideal conditions for nested loops (perfectly nested loops with compile-time constant bounds), we can lose a lot of opportunities for latency-hiding. Consider, for example, a simple two-dimensional loop containing an access to a two-dimensional array as shown by the code in Figure 5.2(a). We will refer to the outer loop in this example as the *i-loop* and the inner loop as the *j-loop*. Constructing a software pipeline to prefetch the data accesses around the inner j-loop yields the code shown in Figure 5.2(b)[1]. Suppose that the number of iterations of the j-loop, `Nj`, is only slightly larger than the calculated prefetch distance, `d`. In this case, we will execute very few iterations in the steady-state. Nearly all of the prefetches will be issued in the prolog section to fill the pipeline, and nearly all of the data accesses will occur in the epilog section to drain the pipeline. Further, because the j-loop is nested inside the outer i-loop, the cycle of filling and draining the pipeline is repeated on each iteration of the outer loop. Figure 5.2(c) depicts the effect of this repeated fill and drain cycle in terms of the number of prefetched pages in the pipeline for the case where `Nj` is slightly larger than `d`. In this situation, very little time is spent in the steady-state where prefetches are effectively overlapped with data accesses.

In general, we expect that `Nj` and `d` will be symbolic values, and thus we cannot know whether `Nj` will be larger than `d` or not at compile-time. If the j-loop is not large enough to hide all the latency, then the steady-state is never reached at all. In this case, the i-loop would have been a better choice for building our pipeline. On the other hand, if `Nj` is much larger than `d`, then the cost of filling the pipeline will be small relative to the time spent in the steady-state and the i-loop would be a poor choice for pipelining. Worse, if the i-loop is not localized (as may occur if `Nj` is extremely large), then constructing the prefetch pipeline across the i-loop would be disastrous. Clearly, there is no single choice of loop that will be right for all circumstances. Instead, we need to take all the loop levels into account and avoid draining the pipeline unnecessarily.

[1]For simplicity, throughout this chapter we do not show the effect of the strip mining optimization, which reduces the frequency of prefetch requests. Our implementation, however, works in the presence of strip-mining as well.

(a) Simple two-dimensional array accesses

```
for (i = 0; i < Ni; i++)
  for (j = 0; j < Nj; j++)
    access(a[i][j]);
```

(b) Software pipelining prefetches around inner loop (the calculation of the prefetch distance, d, is not shown)

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

prolog  for (jprolog = 0; jprolog < min(d, Nj); jprolog++)
          prefetch(&a[i][jprolog]);

steady  j_sw_pipe_upperbound = max(0, Nj-d);
state   for (j = 0; j < j_sw_pipe_upperbound; j++) {
          prefetch(&a[i][j+d]);
          access(a[i][j]);
        }

epilog  for (j = j_sw_pipe_upperbound; j < Nj; j++)
          access(a[i][j]);

    }
```

(c) Effect of repeatedly filling and draining the pipeline

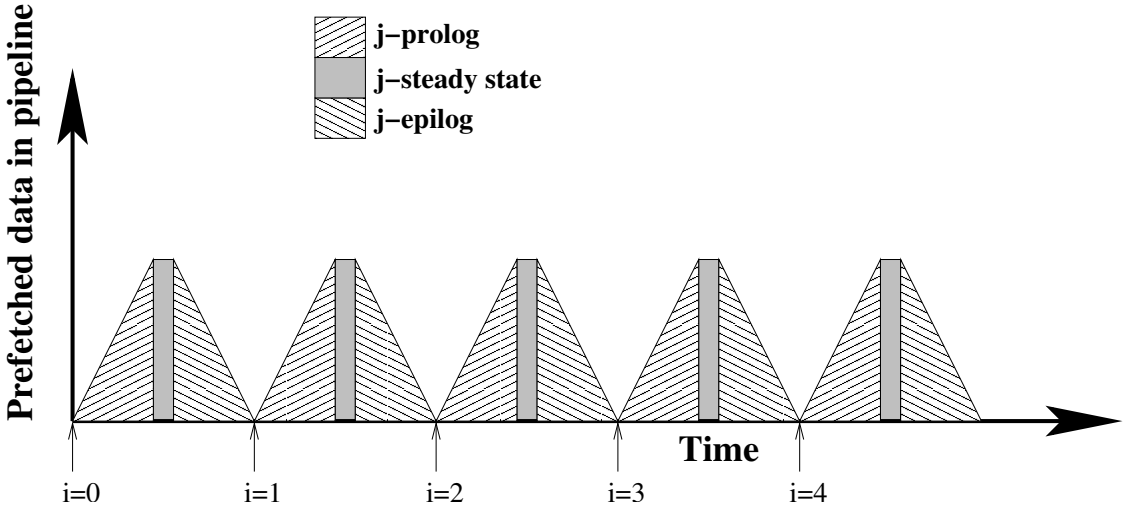


Figure 5.2. Adding prefetches for two-dimensional array accesses

5.2 Developing the Continuous Software Pipelining Algorithm

In this section we consider a sequence of techniques that address various issues with multi-dimensional loops, leading up to the full *continuous software pipelining* algorithm. Throughout, we will focus on the two-dimensional case, although the solutions generalize to higher dimensions.

5.2.1 Calculating prefetch distances for multiple loop nests

When a single loop is chosen as the target for software pipelining, we can express the prefetch distance, d, in terms of iterations of the pipeline loop. If we wish to use multiple loop nests for pipelining, however, we must determine how much of the total page fetch latency should be hidden by each level of nesting.

Consider again the case of a two-dimensional array being accessed within a two-dimensional set of loops, as shown in Figure 5.2(a). When we access an element, `a[i][j]`, in the steady-state of our software pipeline, we want to prefetch the element that will be accessed d iterations of the inner loop in the future. If we only consider the inner loop, then we could prefetch `a[i][j+d]`, as shown in Figure 5.2(b). Depending on the length of the inner loop relative to the prefetch distance, this strategy may or may not be successful at hiding the latency. However, there may be enough work to hide the latency if we take the surrounding loop into account as well by calculating the prefetch address as a function of both loop indices. Each iteration of the outer loop contains N_j iterations of the inner loop, thus to prefetch d iterations of the inner loop ahead, we will need $d_i = d \operatorname{div} N_j$ iterations of the outer loop, leaving $d_j = d \bmod N_j$ iterations of the inner loop. The prefetch address (for the reference of Figure 5.2(a)) is then simply `a[i+`d_i`][j+`d_j`]`.

The general idea is to create an offset variable, d_n, for each loop nest n surrounding the reference to be prefetched. In the steady-state, each loop index variable, i_n, used in an array reference is replaced with $i_n + d_n$ in the corresponding prefetch for that reference. Rather than first calculating a distance in terms of iterations of the innermost loop, we instead calculate how much latency can be hidden by each level of nesting from outermost to innermost based on the length of the loop body at each level, s_n. For loops with symbolic bounds, an expression for s_n is generated by the compiler and evaluated at run-time when the bounds are known. After all the loop offsets have been calculated, any leftover latency is accounted for by increasing the

```
algorithm Compute_Nested_Prefetch_Distances
  latency_left := initial_latency_estimate;
  for n := outermost loop that may be localized to innermost loop do
    d_n := latency_left div s_n; /* s_n is length of body of loop n*/
    latency_left := latency_left mod s_n;
  end for
  if (latency_left > 0) then
    d_innermost := d_innermost + 1;
  end if
end algorithm
```

Figure 5.3. Algorithm for calculating prefetch distances in nested loops.

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

prolog
  for (iprolog = i; iprolog < min(i+di, Ni); iprolog++)
     for (jprolog = 0; jprolog < min(dj, Nj); jprolog++)
        prefetch(&a[iprolog][jprolog]);

steady state
  j_sw_pipe_upperbound = max(0, Nj-d);
  for (j = 0; j < j_sw_pipe_upperbound; j++) {
     prefetch(&a[i+di][j+dj]);
     access(a[i][j]);
  }

epilog
  for (j = j_sw_pipe_upperbound; j < Nj; j++)
     access(a[i][j]);

}
```

Figure 5.4. Sample code for software pipelining a single loop with multiple loop index offsets

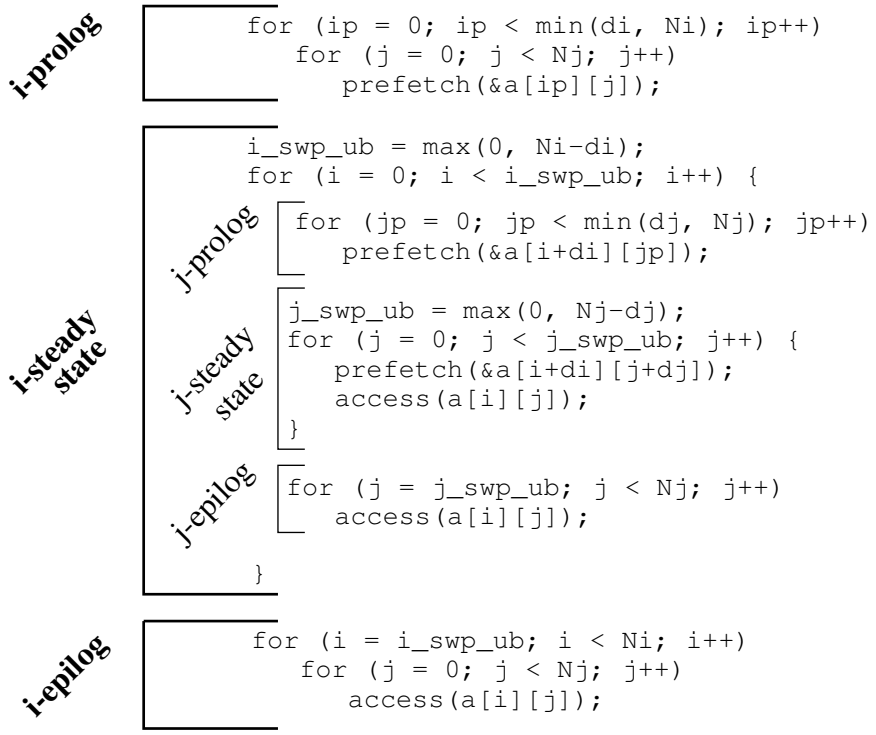
offset for the innermost loop by one iteration.

There is one additional complication that results from the fact that the surrounding loops may not be localized. Pipelining around a loop that is not localized is ineffective (and in fact harmful) since a single iteration of the loop will access enough data to flush any pages prefetched for use in future iterations from memory. In many cases, we are not certain whether a given loop nest is localized or not at compile-time. If, however, a loop is known to be *not* localized, we do not attempt to use that nest or any outer nests in our new pipelining calculations. Figure 5.3 shows the algorithm for computing the loop offsets for an arbitrary set of nested loops.

5.2.2 Nested pipelines

As a first attempt at taking multiple loop nests into account while software pipelining, we could simply choose a single pipelining loop as before, but use the prefetch distances for each surrounding loop nest in

(a) Sample code for nested software pipelines



(b) Effect of nested pipelining on prefetched data

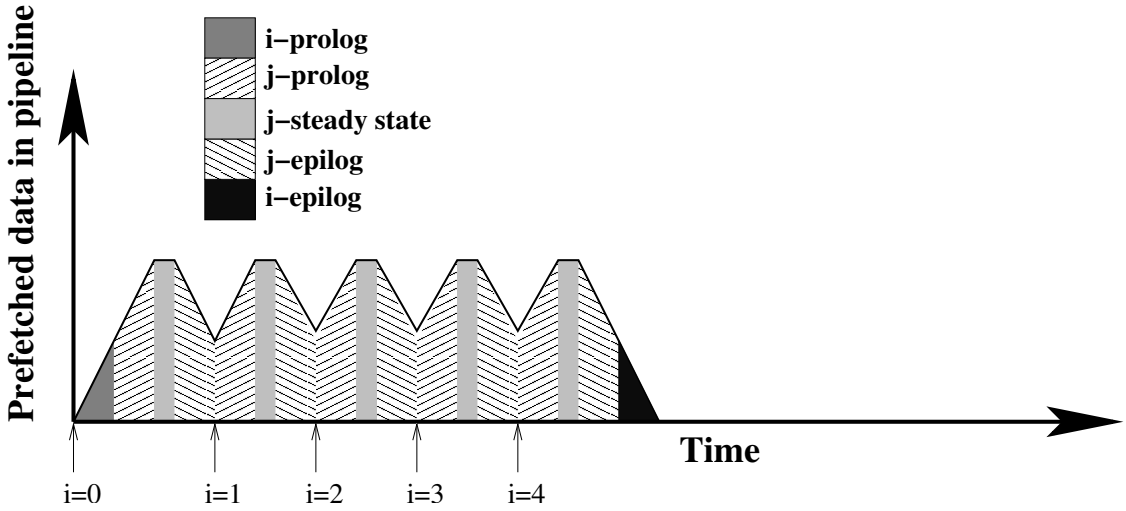


Figure 5.5. Nested software pipelines for two-dimensional array prefetches

the calculation of the prefetch address. Doing so gives the code shown in Figure 5.4. In the prolog, we must prefetch everything between the first item accessed in the steady-state (`a[i][0]`) and the first item prefetched in the steady-state (`a[i+di][j+dj]`). Although this code does use both loop nests to issue prefetches sufficiently far ahead of the reference, it also issues far too many unnecessary prefetches. For every iteration of the outer loop after the first one, in the prolog we repeat the prefetch of some of the same elements that we prefetched on the previous iteration. This is caused by the fact that the first time we see the loop we need to issue a large number of prefetches to fill the pipeline, but on subsequent iterations of the outer loop, we only need to re-fill the portion that was drained during the epilog of the inner loop.

The logical solution is to pull out the portion of the prolog that issues prefetches for the outer loop offset (d_i), placing it before the first iteration of the outer loop. Also, it would be useful to have an epilog for the outer loop as well, to avoid prefetching more than necessary along the outer dimension. The resulting code has the appearance of *nested software pipelines*, as shown in Figure 5.5(a). Around each loop nest we have constructed a pipeline to handle address offsets involving that loop's index variable. Viewed in terms of the effect on prefetched data, the inner epilog only partially drains the pipeline, allowing more of the latency to be hidden, as depicted in Figure 5.5(b). First, prefetches are issued from the outer prolog, then we have a repeated pattern of filling the inner pipeline, executing the inner steady state, and draining the inner pipeline. The outer pipeline is not drained until we get to the end of the pair of loops, which is the real end of the data access. Implementing nested pipelining is straightforward once the appropriate offsets have been calculated for each loop nest - we simply apply the original software pipelining algorithm to each level of the nested loop.

The technique of nested pipelines can be applied to an arbitrary number of dimensions. For n dimensions, we will have $n + 1$ copies of the original loop body. Each loop nest adds one extra copy in its epilog, plus there is one copy in the innermost steady-state. This technique is similar to the *hierarchical reduction* strategy introduced by Lam [35], which allows large units of code (including conditional statements or loop nests) to be treated as a single node for the purpose of software pipelining. Although it is possible to construct examples involving any number of loop nests, all of the concepts we need to illustrate can be shown with

only two levels, so we will focus on the two-dimensional problem in the remainder of this discussion.

At this point we have solved the problem of generating the correct prefetch address when the distance required to hide the I/O latency spans an arbitrary number of loop nests. However, we can still do a better of job of scheduling the prefetches. Figure 5.5(b) shows a general view of the effect of nesting the software pipelines, but the actual effectiveness depends on the amount of latency that we are attempting to hide with each of the nested pipelines. Consider for instance the original two-dimensional loop from Figure 5.2(a). For this example, s_i (the length in cycles of the body of loop i) is equal to $N_j * s_j$. Using the algorithm in Figure 5.3 to calculate the offsets d_i and d_j could lead to many possible situations, depending on the value of N_j and the initial latency estimate. In the worst case, s_i is only slightly larger than the latency estimate and $d_i = 0$. All of the latency will be hidden by the inner pipeline using the d_j offset, however, very little time will be spent in the steady-state and we still pay a large cost for repeatedly filling and draining this pipeline on each iteration of the outer loop. As the latency estimate becomes larger than s_i (either because N_j is smaller or the latency is larger), the outer pipeline comes into play with $d_i > 0$. The larger d_i becomes, the more latency is hidden using the outer pipeline and the smaller the cost of draining the inner pipeline on each iteration.

Effect of Nested Pipelines

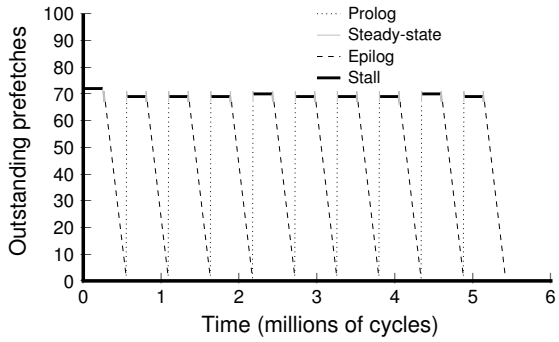
We have created a simple event-driven disk simulator to evaluate our changes to the scheduling algorithm. The simulator models exact LRU replacement, and submits a request to an array of disks on an access or prefetch of a page that is not "in memory". Enough disks are used so that contention does not occur, and we assume the bandwidth between disks and memory is unlimited. Multiple prefetches can proceed in parallel, but accesses must stall until the disk read for the requested page completes. To better isolate the effect of different scheduling algorithms, each innermost loop iteration is assumed to take one cycle to execute, and the disk latency is expressed as a number of cycles (or equivalently, a number of inner loop iterations). Results from the simulator indicate the improvements that could be expected from changes to the scheduling algorithm alone, in the absence of bandwidth limitations, disk contention, latency variations, re-

placement decisions, and other run-time factors that may affect the actual performance on a real system. For our microbenchmark experiments, pages are not reused and thus the replacement algorithm is not a factor.

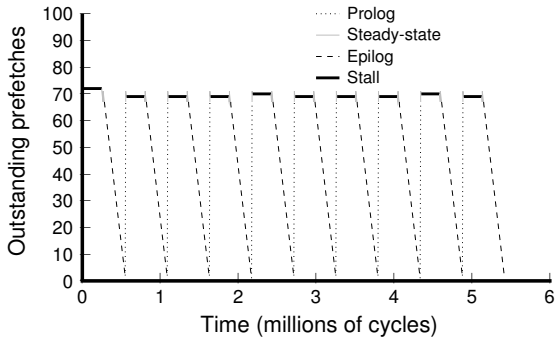
Figure 5.6 compares the original, single-loop pipelining algorithm with the new nested pipelining algorithm for our two-dimensional loop example using three different ratios of s_i to latency. For these examples, our loops access a total of 428 pages of data. Using our event-driven simulator, we record the number of prefetched pages that have not yet been referenced (the *outstanding prefetches*) as a function of simulated time. The number of outstanding prefetches is an indication of the state of the software pipeline, however, this number alone does not tell us anything about whether the right pages have been prefetched, or whether the prefetches were early enough to be useful. To show the effectiveness of the prefetches, we plot stall time as a dark solid line on the graph—better scheduling algorithms will both maintain a full pipeline of outstanding prefetches, and show fewer and shorter stalls, resulting in fewer execution cycles overall.

In the first case, Figure 5.6(a) and (b), we demonstrate the situation where s_i is slightly larger than the latency. On the left (labeled "Original") we show the result for the original pipelining algorithm. The ten iterations of the outer loop are clearly visible as the pipeline of outstanding prefetches is initialized and drained each time. Note that there is a substantial stall following the prolog as each iteration enters the steady-state. This stall occurs because the prolog prefetch of the first page of data has not had time to complete when that page is referenced in the steady-state. Following this stall, there is a very short steady-state phase followed by a long epilog to access all the pages that were prefetched. Beside this graph on the right (labeled "Nested, $d_i = 0$") we show the result of applying the new nested pipelining algorithm in the same situation. The curves are identical to the original case because the inner loop has enough iterations to hide the latency and the outer pipeline is not used. In this situation, we derive no benefit from applying the new nested pipelining algorithm.

In the second case, Figure 5.6(c) and (d), we compare the two algorithms when there are only enough iterations in the inner loop to hide two thirds of the latency. In this case, two thirds of the latency can be hidden using one iteration of the outer loop, and the remaining third can be hidden using half the iterations of the inner loop. On the left, the graph for the original algorithm shows that no time is spent in the steady-

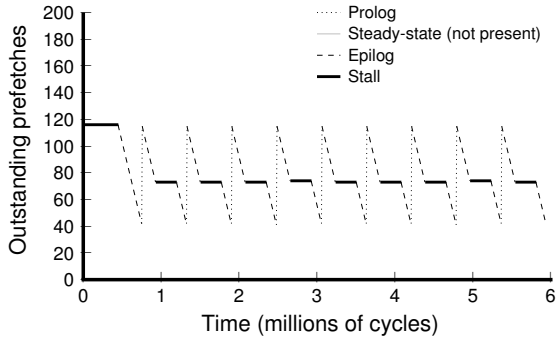


(a) Original

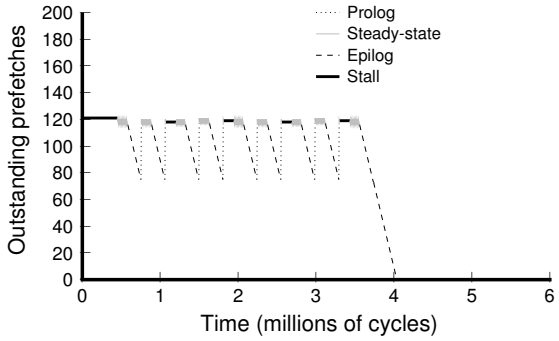


(b) Nested, $d_i = 0$

(a), (b) latency $<$ total time in inner loop

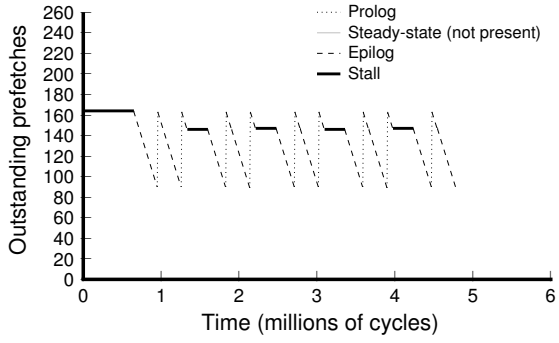


(c) Original

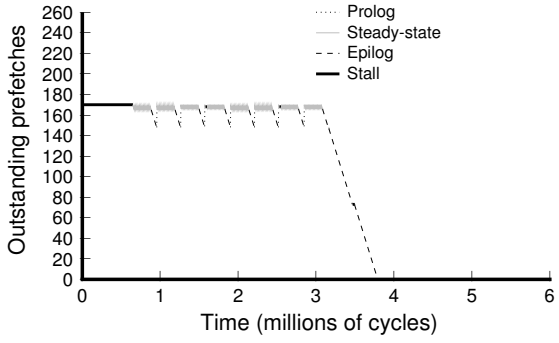


(d) Nested, $d_i = 1$

(c), (d) latency 1.5X total time in inner loop



(e) Original



(f) Nested, $d_i = 2$

(e), (f) latency between 2X and 3X total time in inner loop

Figure 5.6. Comparison of nested pipelining vs. original single-loop pipelining for different latency values

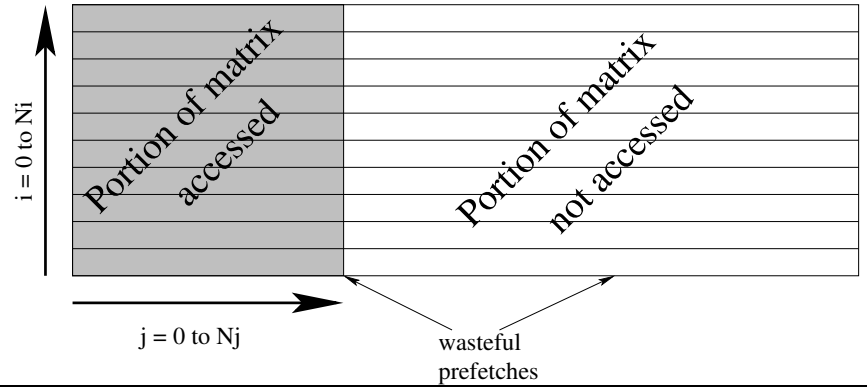


Figure 5.7. Non-sequential data accesses across outer loop: only the first part of each row of matrix is used

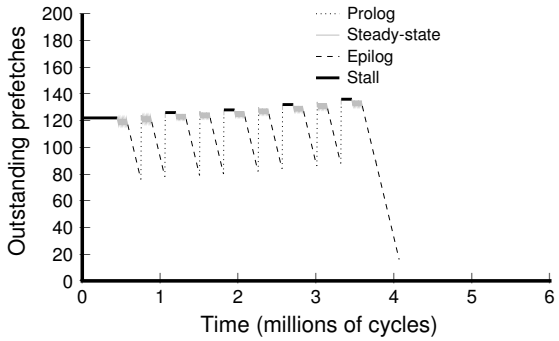
state at all—the code does not use the outer loop, and thus transitions directly from filling the pipeline in the prolog to draining it in the epilog. On the right, however, we see that the nested pipelines are now having a positive effect because they are able to use the outer loop. More time is spent in the steady-state, the pipeline is only partially drained on each outer loop iteration, and the execution time is reduced by nearly one third (compared to the original algorithm in Figure 5.6(c)) due to better overlap of prefetches with computation.

Finally, Figure 5.6(e) and (f) show the result of using the two algorithms when at least two iterations of the outer loop are needed to hide the latency. On the left, the original algorithm shows similar behavior as in (c), with no time spent in the steady-state. On the right, however, we see that the nested pipelining algorithm continues to grow more effective as the latency increases (or, equivalently, as the inner loop gets smaller), with an even larger portion of time spent in the steady-state and reduced penalties for draining the inner pipeline. Somewhat surprisingly, however, nested pipelining only reduces the total execution by about one fifth as compared with the original algorithm for this latency. To explain this unexpected result, note that the original algorithm only stalls five times in this case (vs. ten times in the other two), and with the exception of the first stall, the penalty is partially hidden. This happens because the data accesses are sequential in the inner loop and the compiler-generated code optimistically issues a block prefetch for as many pages as are needed to hide the latency, regardless of how many pages will actually be used in the inner loop. Because the access is also sequential across the outer loop, the extra pages fetched are accessed on subsequent iterations of the outer loop. In this case, the prolog prefetches cover more than two full outer iterations of data, and thus a partial stall only occurs on every second outer loop iteration.

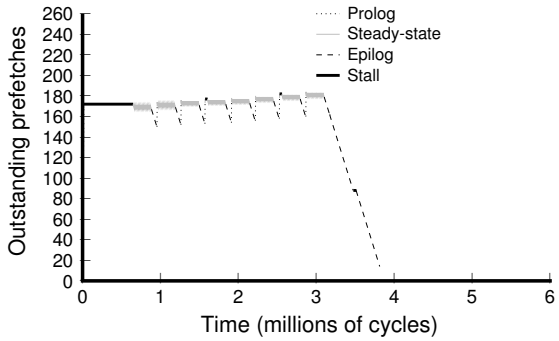
Although issuing large block prefetches in the prolog has the potential to improve performance, there is also a significant potential cost. Note that in Figure 5.6(c) and (e) the number of outstanding prefetches never goes completely back down to zero. That is, the pipeline is never fully drained. For this example, only the extra pages prefetched on the final prolog iteration are wasted. Suppose that the data access was not sequential across the outer loop, however. For instance, suppose the inner loop accesses only the first portion of each row of the matrix as depicted in Figure 5.7. In this case, the extra pages fetched by the original algorithm are not needed; disk bandwidth is wasted and the memory may become polluted. If we take the direct approach and limit the number of pages prefetched to the number that will actually be used in the inner loop, we can solve the problem of wasteful prefetches. Since block prefetches are only used for references with spatial locality, calculating the number of sequential pages that will be referenced in the loop is straightforward[2]. Figure 5.8 illustrates these cases using data from our simple disk simulator.

Figures 5.8(a) and (b) are nearly identical to Figures 5.6(d) and (f), showing that the nested pipelining is able to handle the non-sequential data layout gracefully regardless of the latency. Although there are some wasted prefetches, the number is very small (fewer than 20 pages in both cases, or less than 5% of the number of pages accessed by the test loops). The results for the original pipelining algorithm, depicted in Figures 5.8(c) and (d) provide a sharp contrast. Because the data layout is no longer sequential, the excess prolog prefetches are wasted on each iteration. The larger the latency, the worse this problem becomes—by the end of the simulation, approximately 900 unnecessary pages have been brought into memory for the largest latency simulated. This represents more than twice as many pages as are accessed by the test loops themselves, and would result in extra memory pressure and wasted disk bandwidth in a real system. In addition, because the wrong pages are being fetched, there is no longer a latency-hiding benefit to issuing these large blocks of prefetches. The program still stalls on each iteration of the outer loop. The results of limiting the size of the block prefetches in the original algorithm, to reduce the wasteful prefetches, are shown in Figure 5.8(e) and (f). The shape of these two curves are identical because both are limited to prefetching the number of pages that will be used in the inner loop. Note that this version of the original

[2]We perform this calculation already to avoid releasing more pages than were actually referenced at the end of a loop.

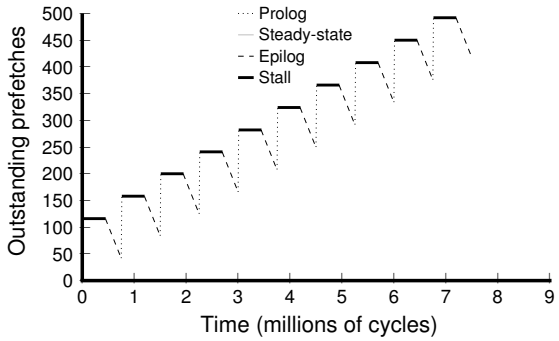


(a) latency 1.5X time in inner loop ($d_1 = 1$)

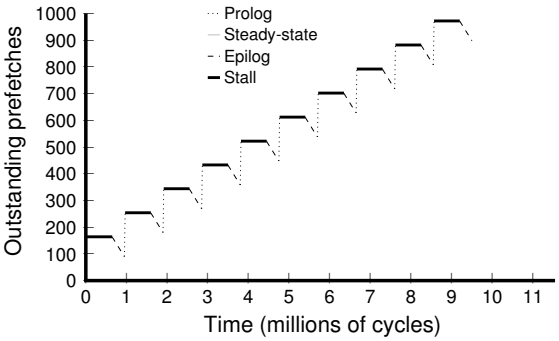


(b) latency >2X time in inner loop ($d_i = 2$)

(a), (b) Nested pipelines with non-sequential data access

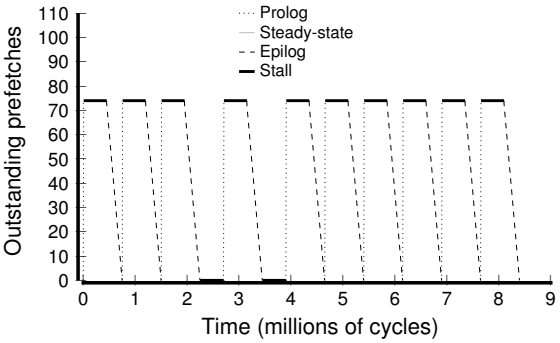


(c) latency 1.5X time in inner loop

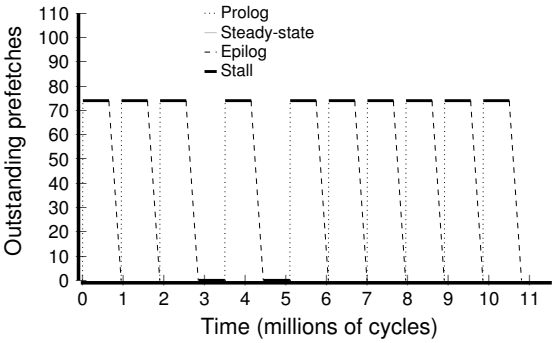


(d) latency >2X time in inner loop

(c), (d) Original pipelines with non-sequential data access



(e) latency 1.5X time in inner loop



(f) latency >2X time in inner loop

(e), (f) Original pipelines, limited prolog prefetches, with non-sequential data access,

Figure 5.8. Effect of non-sequential data access across outer loop

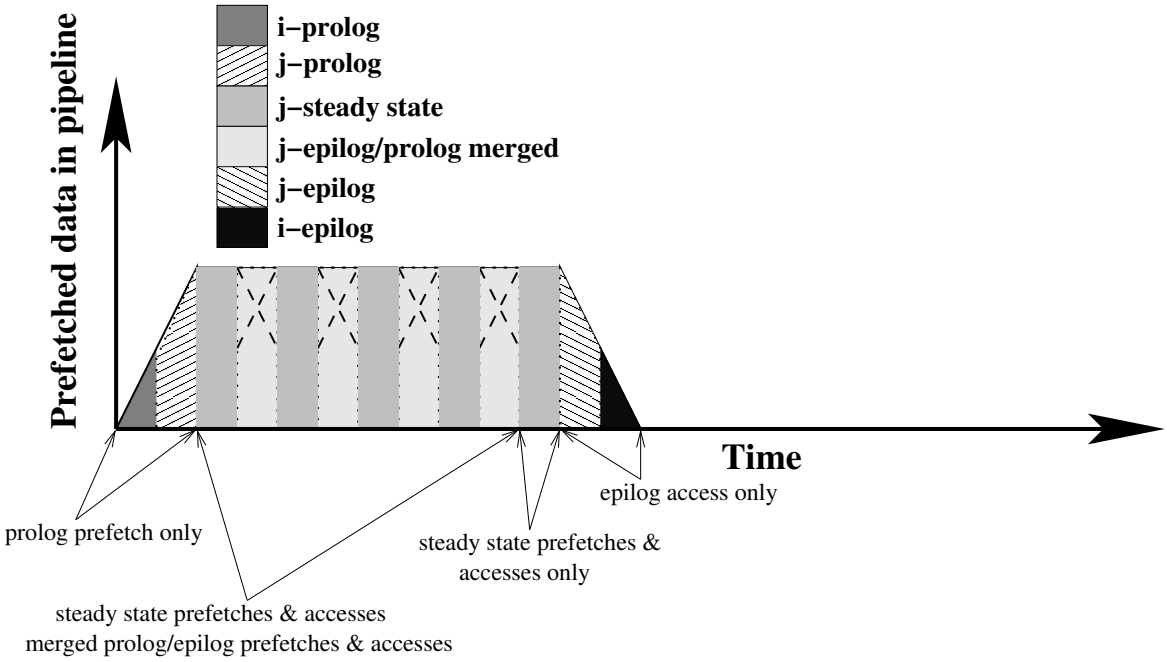


Figure 5.9. Continuous software pipelines for two-dimensional array prefetches

algorithm occasionally stalls at the end of the epilog due to alignment issues that were previously hidden by the excessively-large block prefetches.

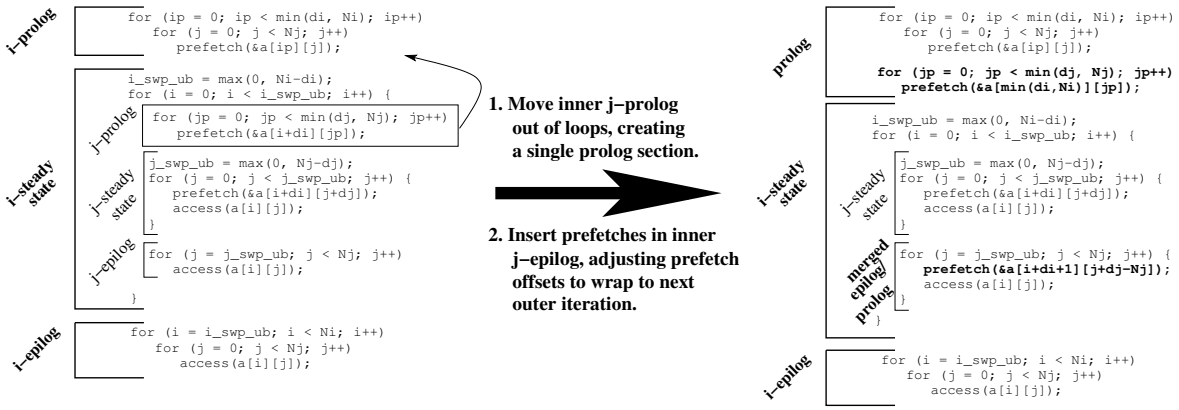
From the graphs in Figures 5.6 and 5.8, it is clear that the nested pipelining algorithm provides significant benefits (both in terms of execution time and better memory usage) when the latency is too large to be hidden by the inner loop. Using memory efficiently is especially important in a multiprogrammed environment where we are also concerned about the negative effects of prefetching on other programs. If the inner loop is large enough to hide the latency, however, there is no benefit to using the nested pipelining algorithm even though a large penalty may still occur due to repeatedly initializing the pipeline, as shown in Figure 5.6(b). We are only better off using nested pipelining if multiple loop nests are actually needed to hide all the latency, but we will typically not know whether or not this is the case until run-time. To solve this problem, we need to start re-filling the pipeline with data from the prolog of the next iteration as we drain out the data in the current epilog. We now discuss how this effect can be achieved by *merging* epilogs with the subsequent prologs.

5.2.3 Merging prologs and epilogs

Figure 5.9 depicts the effect of merging the epilog of one iteration with the prolog of the following iteration. The dashed lines correspond to the epilogs and prologs from the nested pipelining case that have now been merged together, keeping the total number of prefetches in the pipeline steady. There is now a single "fill" stage when we are unable to hide all of the I/O latency, but for most of the execution we are able to stay in the steady-state.

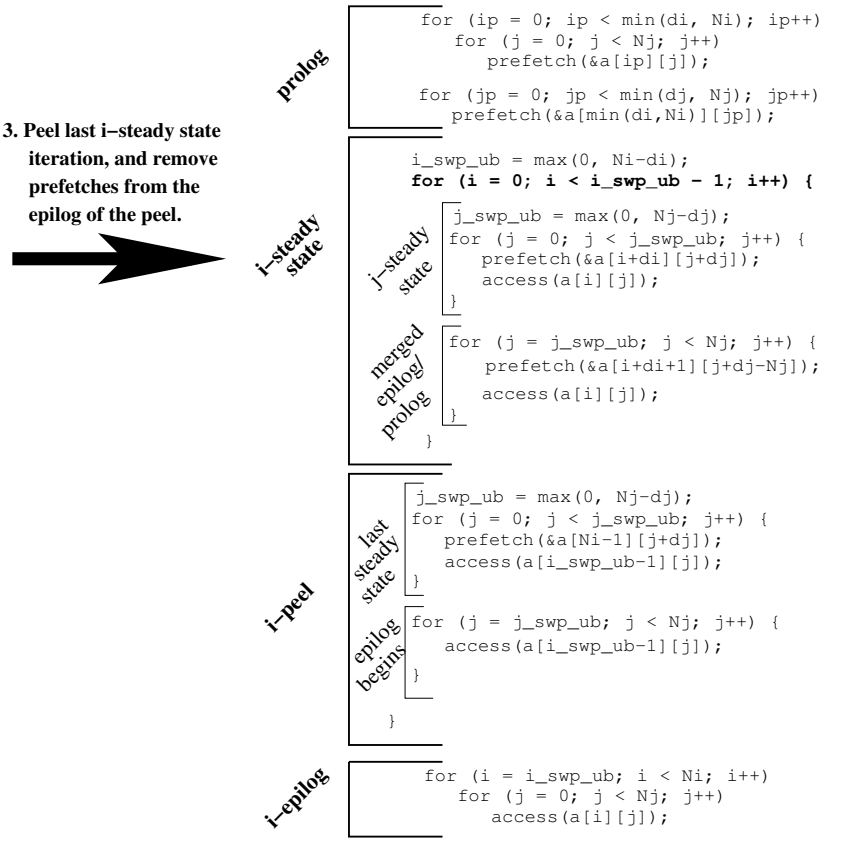
To build a software pipeline that will have the behavior shown in Figure 5.9 we need to do three things to the nested software pipelines. These steps are depicted in Figure 5.10, which shows the transformation from nested to continuous pipelines. First, since the inner prolog and epilog are being merged, we can now pull the first instance of the inner prolog out of the loops, making it part of the outer prolog. Second, in the inner epilog we need to insert the prefetches that would be issued in the inner prolog of the next outer iteration. Normally, the epilog begins at the point where all data that will be accessed by the current loop has been prefetched. This occurs when the loop index plus the prefetch offset reach the original upper bound of the loop. Continuing to prefetch at this point would cause the array index used to generate the prefetch address to *overflow* the range of data originally accessed. Although prefetches are designed so that this overflow is safe (that is, it will not cause an access exception), it may pollute memory by fetching unnecessary data and cannot generally be relied upon to fetch the data that will be needed by the next surrounding loop nest. Rather than overflowing, we want the prefetch index to wrap around to the next surrounding dimension. Clearly this implies that we need to adjust the offsets that are used for prefetching in the epilog. To do so, we simply need to calculate the address that would be used for the prolog prefetch of the surrounding loop, where the outer loop index variable has increased by the loop step and the inner index variable has been reset to its lower bound. For this example, the address to prefetch would be `a[i+di+1][j+dj-Nj]`. The result of these two steps produces the code shown in Figure 5.10(b).

Merging the epilog with the next iteration's prolog is successful until the point where wrapping the index for the inner loop causes the prefetch index for the outer loop to overflow. This happens during the final



(a) 2–D nested pipelining example

(b) Result of first 2 steps in transformation to continuous pipelining



(c) Final 2–D continuous pipelining example

Figure 5.10. Progression from Nested pipelines to Continuous pipelines

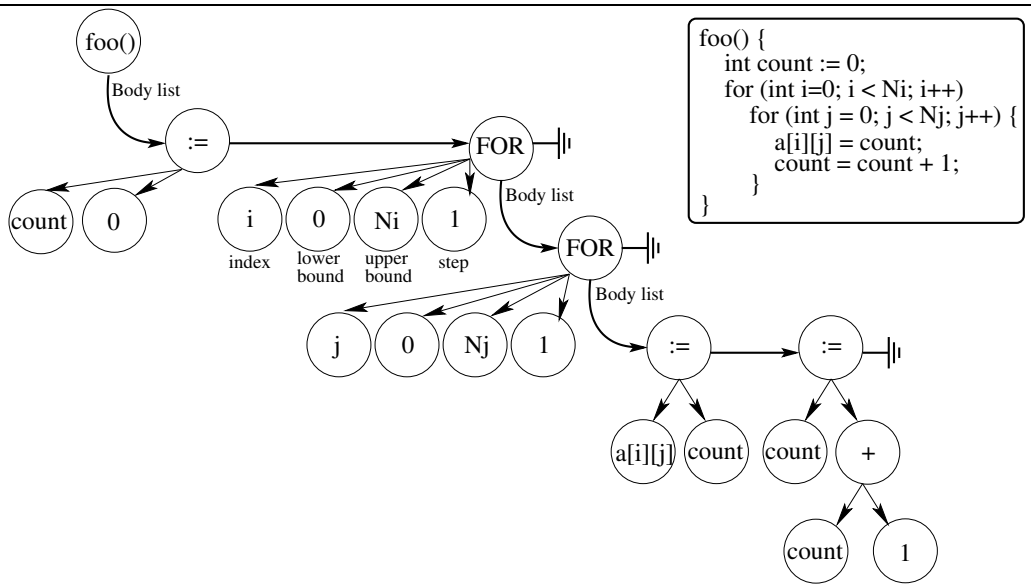


Figure 5.11. SUIF-style abstract syntax tree for simple procedure with loops

iteration of the outer steady-state. Ordinarily, the code would have switched to the epilog at the point where the prefetch index overflowed, but the overflow now occurs one iteration earlier due to the wrapping of the inner index. In this final outer steady-state iteration, we still want to issue the inner steady-state prefetches, but we don't have another prolog to merge into the epilog. The third step is thus to isolate the final iteration of the steady-state by peeling it, continuing to prefetch only in the inner steady-state of the peel. In the inner epilog of the peel we remove the prefetch instructions. The effect of this final step is shown in Figure 5.10(c). This code has all of the properties we want for I/O prefetching: the prefetch addresses are calculated correctly to hide large latencies, and the software pipeline remains filled as long as there is more data that needs to be prefetched.

Having informally introduced the strategy for building continuous software pipelines, we turn now to a detailed description of the algorithm. For concreteness, we assume the algorithm operates on program code in a tree-structured intermediate format such as that used by SUIF. In this representation, procedures are the roots of abstract syntax trees. Each procedure contains a list of tree nodes (or simply *nodes* in this discussion) which form the procedure body. For our purposes, we are primarily interested in FOR-nodes, which are used to represent for-loops. A FOR-node contains an index variable, lower bound, upper bound and step expressions, and a list of nodes that form the body of the loop. Figure 5.11 illustrates the basic structure of a

Properties of a set of nested loops, N	
$innermost[N]$	The innermost loop in the nest
$outermost[N]$	The outermost loop in the nest
Properties of a loop, L	
$body[L]$	A list of nodes forming the current body of the loop L
$original\text{-}body[L]$	The original body of L before transformations
$lower\text{-}bound[L]$	The lower bound of loop L
$upper\text{-}bound[L]$	The upper bound of loop L
$index[L]$	The index variable of loop L
$step[L]$	The amount $index[L]$ is incremented on each iteration of loop L
$outer[L]$	The loop immediately surrounding L, i.e., the next outer loop from L
$prefetch\text{-}offset[L]$	The number of iterations to pipeline loop L for prefetching
Properties of nodes and lists of nodes	
$head[node\text{-}list]$	The first node in the list
$next[node]$	The successor of $node$, i.e., the next node in the list

Table 5.2. Properties used in presenting continuous pipelining algorithm

COPY-LOOP-BODY($loop$)	Creates a complete copy of $body[loop]$
BUILD-ORIGINAL-PIPELINE($loop$)	Constructs a standard software pipeline around $loop$ using the $prefetch\text{-}offset[loop]$
INSERT-BEFORE($node$, $position$)	Inserts $node$ into the list containing $position$, immediately preceding $position$
INSERT-PREFETCHES($loop$, $nest$)	for each array reference in $loop$, adds a prefetch instruction to $body[loop]$, replacing each use of $index[loop]$ in the array index expression with ($index[loop]$ + $prefetch\text{-}offset[loop]$)
ADD-ANNOTATION($loop$, $mesg$)	Attaches a simple annotation,$mesg$, to $loop$
PEEL-LAST-ITERATION($loop$)	Peels the last iteration of $loop$ and returns the resulting list of nodes

Table 5.3. Additional procedures used in presenting continuous pipelining algorithm

simple procedure containing two nested for-loops in this representation (details such as the symbol table and the termination test on the for-loop are omitted to focus on the tree structure).

In presenting the algorithm itself, we adopt the pseudocode style of Cormen et al. [18]: indentation indicates block structure, the "▷" symbol indicates a comment, and data objects are described by attributes, written as $attribute[object]$. We will refer to three main types of objects: sets of nested loops (or *nests*), for-loop nodes (or *loops*), and lists of arbitrary nodes. Table 5.2 summarizes the attributes of these objects which we will be using. Finally, the pseudocode for the main algorithm uses several additional procedures which we describe briefly in Table 5.3.

Our algorithm, shown in Figure 5.12, needs to handle separately the cases of the innermost loop (lines 9–13) and surrounding loops (lines 15–29), which include intermediate loops (lines 20–23) and the outermost loop (lines 25–28). We begin by generating a copy of the original bodies of all of the loops (lines 1–2).

This simplifies the construction of the final epilog code where no prefetching transformations are needed. Although we need only a single prolog section at the beginning of all the loops for continuous pipelining, we will build the prolog incrementally as each loop is pipelined. We thus mark the point where the prolog should be added (line 3) and then proceed to transform the loops, beginning with the innermost and working out to the outermost loop (lines 4–29).

For each loop, we start by generating $prolog$, $steady$ and $epilog$ loops according to the original pipelining algorithm; we use the $prefetch\text{-}offset$ previously calculated to hide the required portion of the latency with the current loop (line 5). The prolog is inserted immediately before the previous prolog, or before the outermost loop in the case of the first prolog generated (lines 6–7). This causes the prologs to be ordered from outermost to innermost in the list of nodes preceding the outermost loop. Next we deal with the three types of loops.

Lines 9–13 handle the case of innermost loops. Prefetches are added to the $steady$ and $epilog$ loops using the original $prefetch\text{-}offsets$ for each loop. The prefetch offsets are then adjusted in the $epilog$ loop to cause the address calculation to wrap around to the next iteration of the surrounding loop. An annotation is added to the epilog loop to allow us to identify it later when surrounding loops are pipelined, and the original loop, L is replaced in the body of the surrounding loop with the pair of loop nodes, $steady$ and $epilog$. Note that because this transformation of the innermost loop is performed first, all surrounding loops will contain a copy of the pipelined innermost loop with prefetching code already added to it when they are pipelined.

We next have to deal with the cases of loops surrounding the innermost. For each of these loops, the epilog begins when the prefetch address component for the current loop would overflow. As with the simple two-dimensional example discussed earlier, however, wrapping the offsets of the next inner loop will cause the current loop's prefetch index to overflow one iteration earlier. We thus need to peel the final iteration of the $steady$ loop (line 15). Within the peel, we need to identify the point where the overflow would occur. For intermediate loops, this is the point where we have to begin wrapping the prefetch index for the current loop. For the outermost loop there is no surrounding loop onto which to wrap the index, so the overflow point represents the point where prefetching is no longer needed. Conceptually, we must look within the current peel for the loop where the prefetch index for the next inner loop began wrapping. Precisely how to locate

```
BUILD-CONTINUOUS-PIPELINES(N)
    ▷ Step 1: Preserve copy of original loop bodies
 1  for L ← outermost[N] to innermost[N] do
 2          original-body[L] ← COPY-LOOP-BODY(L)
    ▷ Step 2: Transform Loops
 3  prolog-position ← outermost[N]
 4  for L ← innermost[N] to outermost[N] do
 5          (prolog, steady, epilog) ← BUILD-ORIGINAL-PIPELINE(L, prefetch-offset[L])
 6          INSERT-BEFORE(prolog, prolog-position)
 7          prolog-position ← prolog
 8          if L = innermost[N] then                          ▷ Case 1: L is innermost loop
 9                  INSERT-PREFETCHES(steady, N)
10                  INSERT-PREFETCHES(epilog, N)
11                  ADJUST-PREFETCH-OFFSETS(epilog, L, outer[L])
12                  ADD-ANNOTATION(epilog,"Epilog Loop")
13                  Replace L in parent list with (steady, epilog)
14          else                                              ▷ Case 2 & 3: surrounding loops
15                  peel-list ← PEEL-LAST-ITERATION(steady)
16                  node ← head[peel-list]
17                  while node does not have "Epilog Loop" annotation do
18                          node ← next[node]
                    ▷ node is now loop with "Epilog Loop" annotation
19                  if L ≠ outermost[N] then                  ▷ Case 2: L is intermediate loop
20                          while node ≠ NIL do
21                                  ADJUST-PREFETCH-OFFSETS(node, L, outer[L])
22                                  node ← next[node]
23                          ADJUST-PREFETCH-OFFSETS(epilog, L, outer[L])
24                  else                                      ▷ Case 3: L is outermost loop
25                          while node ≠ NIL do
26                                  Replace body[node] with original-body[node]
27                                  node ← next[node]
28                          Replace body[epilog] with original-body[epilog]
29                  Replace L in parent list with (steady, peel-list, epilog)
```

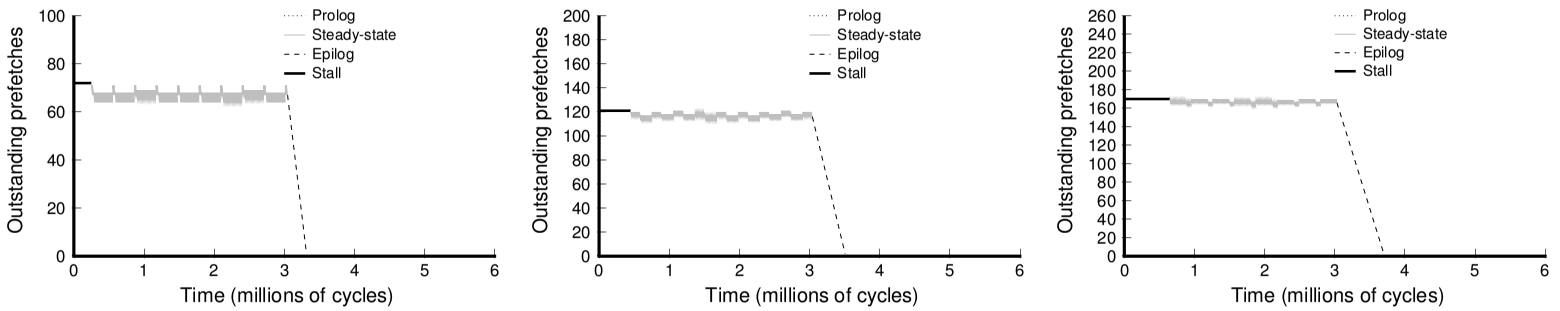
```
ADJUST-PREFETCH-OFFSETS(node, current, outer)
1  if node is not a loop then
2          Return
3  foreach prefetch instruction P in body[node] do
4          foreach expression E containing index[current] in P do
5                  E ← E − upper-bound[current] + lower-bound[current]
6          foreach expression E containing index[outer] in P do
7                  E ← E + step[outer]
```

Figure 5.12. Algorithm for building continuous software pipelines in nested loops

this loop depends on the representation. In our case, we take advantage of the tree structure of the SUIF intermediate format as follows. We observe that wrapping always occurs in the copy of the innermost epilog in the peel of each loop, however an arbitrary interior loop peel will have multiple copies of the innermost epilog. The key is that each peel removes a layer of nesting. We thus require the copy of the innermost epilog that occurs at the top level in the current $peel\text{-}list$. Another way to think of this property is that in the innermost epilog we must wrap the prefetch address onto the next surrounding loop. All copies that are not at the top level in the peel have some other loop surrounding them, which is the target of the wrapping. When the copy is at the top level, however, the overflow condition must be handled by the current loop. To find the point where overflow would occur, we simply walk the list of nodes in the $peel\text{-}list$ looking for the loop with the epilog annotation added earlier (lines 16–18).

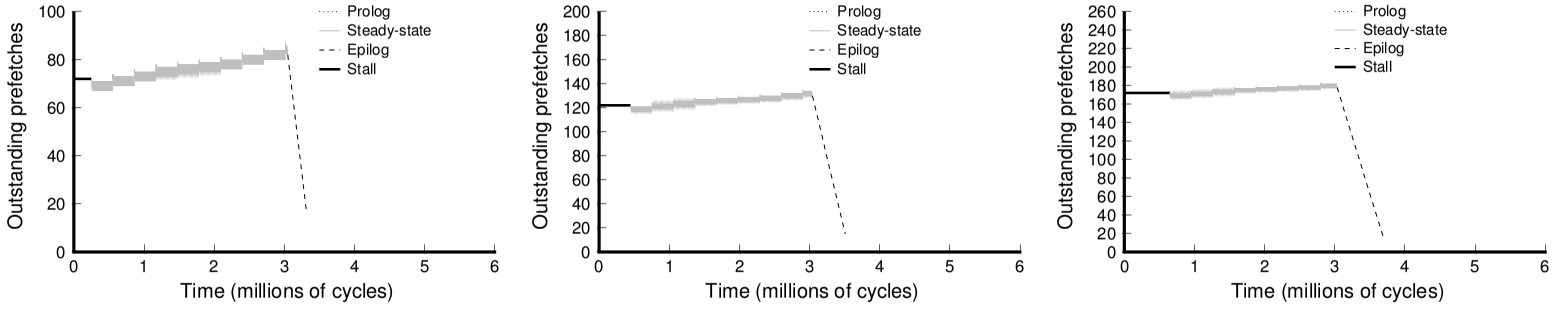
After locating the point where the prefetch addresses would overflow, we need to distinguish between intermediate loops, where we can wrap prefetches to a surrounding loop, and the outermost loop where prefetching should stop. For intermediate loops (lines 20–23) we adjust the prefetch offsets for the current loop and the next outer loop to wrap the prefetch addresses for every prefetch instruction in every loop until the end of the peel list. We also adjust the prefetch addresses in the epilog in the same manner. For the outermost loop, rather than adjusting the prefetch offsets, we simply replace the body of each loop in the rest of the peel, and the body of the epilog with the original, untransformed body of that loop. Finally, after all necessary adjustments have been made, we replace the original loop L with the $steady$ loop, the $peel\text{-}list$ and the $epilog$ loop in the list containing L.

Figure 5.13 illustrates the simulated performance of the continuous pipelining algorithm under the different latency and data layout scenarios introduced previously. In contrast to the corresponding graphs for nested pipelining in Figures 5.6 and 5.8, the continuous pipelining curves are nearly identical in all situations. There is a single stall when the steady-state is first entered, because the first page prefetched has not had time to complete. Thereafter, the number of outstanding prefetched pages remains nearly constant until the sole epilog at the end of the simulation. There is a small amount of "jitter" in the pipeline as transitions are made from the original steady-state to the merged prolog/epilog portion of the steady-state and back, however, these



(a) latency $<$ time in inner loop, $d_i = 0$ (b) latency 1.5X time in inner loop, $d_i = 1$ (c) latency $>$ 2X time in inner loop, $d_i = 2$

(a), (b), (c) Continuous pipelining with sequential data accesses across outer loop



(d) latency $<$ time in inner loop, $d_i = 0$ (e) latency 1.5X time in inner loop, $d_i = 1$ (f) latency $>$ 2X time in inner loop, $d_i = 2$

(d), (e), (f) Continuous pipelining with non-sequential data accesses across outer loop

Figure 5.13. Simulated performance of continuous software pipelining under varying latency and data layouts

variations do not impact on the overall effectiveness of the software pipelining. As with nested pipelining, there are fewer than 20 wasted prefetches (less than 5% of the data accessed) when the data layout is not sequential across outer loop iterations. Regardless of the amount of latency, or the data layout, the continuous pipelining algorithm is able to effectively use multiple loop nests to overlap prefetches with data accesses.

The algorithm in Figure 5.12 is able to handle arbitrary levels of nesting, including imperfectly-nested as well as perfectly-nested loops. In all cases, prefetches are correctly generated to maintain a full software pipeline until all needed data has been prefetched. Scheduling prefetches correctly is not the only issue, however. Referring back to Figure 5.10(c), notice that there are now five copies of the original loop body for the two-dimensional loop nest. In general, the strategy outlined here will lead to $3^{n-1} + n$ copies of the original loop body, for n levels of loops since all loops except the innermost need to distinguish three separate situations. This may lead to an unacceptable expansion in code size for even moderately deep nestings. The important issue is the dynamic code footprint, which will depend on how the execution is split between the various cases in the continuous pipeline based on actual loop bounds. The continuous pipelines are flexible enough to adapt to a large number of situations, but it is unlikely that each case will occur with equal frequency. Nonetheless, we will demonstrate how the code expansion problem can be handled.

5.2.4 Reducing code expansion

It is tempting to address the code expansion problem by ignoring the special case that occurs in the final iteration of the steady-state, and continuing to transition to the epilog at the same point as before. This strategy eliminates the need to peel an iteration of each surrounding loop, and reduces the number of copies to what we had for the basic nested pipelines. Unfortunately, if we do not know the loop bounds, we cannot estimate how important those final steady-state prefetches really are. For example, in our simple two-dimensional example, if `Ni` is only 2, and `Nj` is extremely large, roughly half of the prefetches should be issued in the last steady-state iteration of the outer i-loop. By ignoring this possibility, we greatly reduce our opportunities to hide I/O latency in some cases.

Fortunately, there is a better solution. Observe that all the copies of the original loop body occur in one

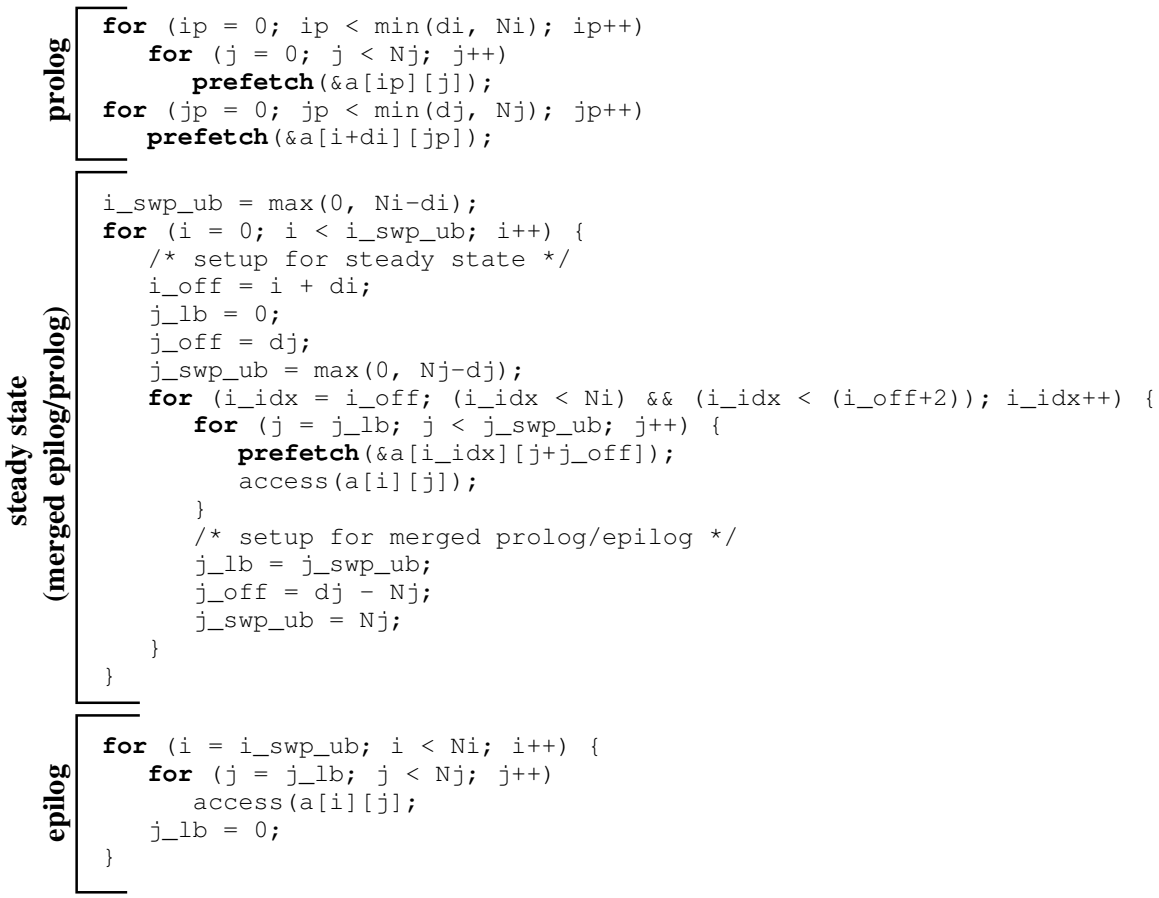


Figure 5.14. Wrapped pipeline: only two copies of the original loop body are needed to cover all situations.

of only two situations: either the original body by itself, or the original body with a prefetch operation added. The only differences are the upper and lower bounds on the loops surrounding the body, and the offsets used in the prefetch address calculation. By identifying the points where these values need to change and inserting code to calculate them appropriately, we only need a single copy of the body with the prefetch operation. Similarly, if we can identify the point where prefetching should stop, we only need a single copy of the epilog code.

Our solution is to "wrap" each inner loop in a new loop that executes at most twice. Before the first iteration of the wrapper we insert code to set the bounds on the "wrapped" loop for the steady state condition. At the end of the wrapper, we insert code to set bounds and offsets for the merged epilog/prolog situation. An example for the two-dimensional case is shown in Figure 5.14. This technique for reducing code expansion

Bench	Version	Static Code Size	Secondary I-cache Misses (x 10^5)	% Increase Over Previous Version	Execution Time (cycles) (x 10^9)	% Increase Over Previous Version
FFTPDE	No Prefetch	10.9K	2.76	N/A	62.36	N/A
	Original pipe	16.1K	3.07	11.2	62.55	0.3
	Dynamic lat.	21.7K	3.19	3.7	62.34	-0.3
	Nested pipe	25.0K	3.31	3.8	62.79	0.7
	Continuous	46.0K	3.84	16.1	63.36	0.9
MGRID	No Prefetch	13.1K	0.58	N/A	23.76	N/A
	Original pipe	23.3K	0.65	11.9	24.24	2.1
	Dynamic lat.	24.4K	0.69	5.0	25.55	5.4
	Nested pipe	39.5K	0.83	21.0	27.40	7.3
	Continuous	144.8K	1.34	61.1	26.94	-1.7
APPBT	No Prefetch	81.2K	1.70	N/A	11.31	N/A
	Original pipe	199.5K	1.85	9.2	11.81	4.4
	Dynamic lat.	203.0K	1.92	3.4	11.82	0.1
	Nested pipe	290.1K	2.58	34.6	11.81	-0.1
	Continuous	845.5K	3.39	31.3	11.87	0.5
APPLU	No Prefetch	58.2K	0.27	N/A	2.69	N/A
	Original pipe	98.5K	0.27	-1.1	2.74	2.0
	Dynamic lat.	100.5K	0.29	6.1	2.74	0.2
	Nested pipe	172.0K	0.39	37.8	2.76	0.4
	Continuous	442.4K	0.61	55.2	2.82	2.5
APPSP	No Prefetch	75.2K	5.62	N/A	59.49	N/A
	Original pipe	133.1K	5.59	-1.5	60.40	1.53
	Dynamic lat.	137.2K	5.84	4.5	60.22	-0.3
	Nested pipe	224.4K	6.21	6.2	62.64	4.0
	Continuous	593.8K	10.96	76.5	63.53	1.4

Table 5.4. Effect of pipelining algorithms on code size and execution time

is similar in spirit to loop rerolling [5], which has also been applied to reduce the memory footprint for embedded applications [1]. The wrapped pipeline approach introduces some additional overhead, in the form of the extra loop and the code to update bounds and offsets.

To evaluate whether this optimization is necessary, we applied the nested and continuous pipelining algorithms to the NAS parallel benchmarks that have nested loops. We then measured the expansion in static code size, the change in i-cache misses and the effect on execution time. To show the worst-case effect of code expansion alone, we disabled the actual prefetch and release operations, and ran these experiments on a quiet system with enough memory to eliminate paging. The results are shown in Table 5.4.

As expected, each enhancement to the scheduling algorithm results in an increase in the static code size, with the most dramatic increase occurring with the continuous pipelines. The dynamic effects are less pronounced and less uniform, however. Looking at the second-level instruction cache misses, we see that the

misses generally increase with larger static code sizes, and the largest increase again occurs for the continuous pipelines. In the worst case, continuous pipelining for APPSP results in a 76% increase in instruction cache misses over the nested pipelining algorithm. We are most interested in the effect of code expansion on the overall execution time, however, and here we see that continuous pipelining has a worst case penalty of just 2.5% over nested pipelining (for APPLU). In the case of APPSP, the massive increase in instruction cache misses leads to only a 1.4% increase in execution cycles. For MGRID, the execution time actually *decreases* with the continuous pipelining algorithm (relative to nested pipelining) in spite of a 61% increase in instruction cache misses. Clearly other factors also have an effect on the overall performance, even in a setting designed to highlight the impact of code expansion.

In a more realistic usage scenario (out-of-core applications with prefetching and releasing occurring), other effects have a far more dramatic impact on instruction cache performance than the choice of software pipelining algorithm. For instance, the main execution thread is less likely to be scheduled on the same CPU when we have active prefetch helper threads, and the execution of operating system code to handle prefetches and page faults both lead to large increases in the number of instruction cache mises.

Based on these experiments, we conclude that the wrapped pipelining algorithm is unlikely to have any benefits relative to the fully-expanded continuous pipelining algorithm for the types of applications we are targeting, and we do not consider it any further in this dissertation. There may, however, be other settings where code footprint is a larger concern.

Having introduced our new algorithms for scheduling prefetches in out-of-core applications, we now discuss an important property of these algorithms—the ability to deal with imperfectly-nested loops.

5.2.5 Imperfectly nested loops

In his thesis, McIntosh [37] also considered compiler algorithms for prefetching in nested loops when latencies were large. Although he was concerned with prefetching cache misses, he observed that inner loops with short trip counts could be too small to effectively hide long-latency memory accesses. McIntosh's solution, termed *outer-loop pipelining*, is similar to our heuristic for finding a suitable loop nest (as described

in Section 3.1.2) when the bounds are known at compile-time. When the bounds are unknown, however, outer-loop pipelining first strip-mines the inner loop and then interchanges the strip-mine loop with the outer loop to give the compiler control over the prefetching distance, which is now expressed in terms of the strip-mining factor. The effectiveness of outer-loop pipelining was found to be mixed, producing modest improvements in some cases, and modest degradations in others [37]. Further, the technique is only applicable to perfectly-nested loops, and the compiler must be able to prove that the interchange is safe in the case of unknown bounds.

In contrast, because we are not attempting to interchange the loops themselves, the continuous pipelining strategy can be applied to imperfectly-nested loops. This feature is important, particularly when considering the very long latencies that must be hidden for I/O prefetching. As the latencies become larger, it is not sufficient to consider only the loop immediately surrounding the innermost one. For example, in the APPBT, APPLU and APPSP benchmarks in the NAS Parallel suite, the most suitable loop for I/O prefetching frequently contains multiple inner loops. Figure 5.15 shows an example drawn from the APPLU benchmark. Consider the "`do 999980`" loop (the i-loop), which contains four distinct loops. Of these four, the two perfectly-nested loops each have a total trip count of only 25 iterations, while the remaining two are themselves imperfectly-nested (and also have short trip counts). For I/O prefetching to have any chance of success, we must at least pipeline the i-loop, which would be impossible using McIntosh's outer loop pipelining strategy.

The major challenge posed by imperfectly-nested loops for continuous pipelining is the possibility that it will be too early to start the next prolog during the current epilog of an inner loop. To illustrate, refer again to Figure 5.15. When we generate the epilog for the first loop nested inside i (the "`do 99988, m = ...`" loop), we insert prefetches for the next iteration of the i-loop, however, we still have to execute the other three loops that are nested inside the i-loop. If these later loops access a large amount of data, it is possible for the data prefetched in the first epilog to be flushed from memory before it can be used. In this specific example, the bounds on all the loops nested inside the i-loop are constant at compile-time, so we can estimate the volume of data accessed and determine whether continuous pipelining will be fruitful or

```
        do 99978, k = 2, nz - 1
          do 99979, j = 2, ny - 1
            do 99980, i = 2, nx - 1
C             ***First loop nested inside i
              do 99988, m = 1, 5
                do 99989, l = 1, 5
                  v(m, i, j, k) = v(m, i, j, k) - omega * (ldz(m, l, i,
     *j, k) * v(l, i, j, k - 1) + ldy(m, l, i, j, k) * v(l, i, j - 1, k)
     * + ldx(m, l, i, j, k) * v(l, i - 1, j, k))
99989           continue
99988         continue
C             ***diagonal block inversion
C             ***forward elimination - Second loop nested inside i
              do 99986, m = 1, 5
                do 99987, l = 1, 5
                  tmat(m, l) = d(m, l, i, j, k)
99987           continue
99986         continue
C             ***Third loop nested inside i
              do 99983, ip = 1, 4
                tmp1 = 1.0d+00 / tmat(ip, ip)
                do 99984, m = ip + 1, 5
                  tmp = tmp1 * tmat(m, ip)
                  do 99985, l = ip + 1, 5
                    tmat(m, l) = tmat(m, l) - tmp * tmat(ip, l)
99985             continue
                  v(m, i, j, k) = v(m, i, j, k) - v(ip, i, j, k) * tmp
99984           continue
99983         continue
C             ***back substitution - Fourth loop nested inside i
              do 99981, m = 5, 1, -1
                do 99982, l = m + 1, 5
                  v(m, i, j, k) = v(m, i, j, k) - tmat(m, l) * v(l, i, j
     *, k)
99982           continue
                v(m, i, j, k) = v(m, i, j, k) / tmat(m, m)
99981         continue
99980       continue
99979     continue
99978   continue
```

Figure 5.15. Example of imperfect loop nesting structure found in blts subroutine of applu.f

Bench	Version	Simulated Time (cycles)	Simulated Stall (cycles)	Page Faults	Pages Prefetched	Pages Replaced Before Use	Pages Not Prefetched	Late Prefetches
FFTPDE	No Prefetch	1.00e12	9.66e11	369,963	N/A	N/A	369,963	N/A
	Original pipe	1.45e11	1.06e11	380,323	378,029	37,608	39,980	86,623
	Nested pipe	7.08e10	3.18e10	373,538	368,479	1,192	6,267	8,635
	Continuous	9.67e10	5.77e10	374,171	365,878	8,765	17,091	5,582
MGRID	No Prefetch	9.36e11	8.54e11	327,038	N/A	N/A	327,038	N/A
	Original pipe	9.87e10	1.60e10	330,784	401,248	70,453	19	28,602
	Nested pipe	1.06e11	2.36e10	327,286	329,604	2,344	28	22,092
	Continuous	9.92e10	1.65e10	327,740	330,917	7,075	3,925	3,729
APPBT	No Prefetch	4.61e11	4.22e11	161,686	N/A	N/A	161,686	N/A
	Original pipe	1.79e11	1.40e11	162,275	121,602	4,406	47,107	6,575
	Nested pipe	7.31e10	3.31e10	163,566	154,197	219	10,123	8,673
	Continuous	6.73e10	2.73e10	162,359	160,187	5,714	9,168	10,625
APPLU	No Prefetch	5.68e11	5.28e11	202,432	N/A	N/A	202,432	N/A
	Original pipe	9.14e10	5.19e10	226,414	1,224,058	1,016,069	18,908	3,657
	Nested pipe	9.96e10	6.03e10	202,846	186,934	1,331	17,324	19,262
	Continuous	1.08e11	6.84e10	203,051	184,204	2,824	21,779	4,862
APPSP	No Prefetch	2.92e12	2.78e12	1,064,552	N/A	N/A	1,064,552	N/A
	Original pipe	8.76e11	7.34e11	1,086,023	1,449,012	421,792	59,842	234,848
	Nested pipe	3.71e11	2.29e11	1,066,079	1,026,117	2,820	43,641	110,240
	Continuous	3.28e11	1.86e11	1,068,317	1,016,433	8,425	61,221	43,479

Table 5.5. Simulated performance characteristics

not. If, however, we cannot estimate the volume of data traffic, we fall back on an all-or-nothing decision, settable with a compile-time flag. The options are to either assume imperfectly-nested loops of unknown size do not access enough data to flush the prefetches from memory, or assume that they do access a large volume of data and use the nested pipelining strategy instead. In our experiments, we compile with the assumption that imperfectly-nested loops will not cause prefetched data to be flushed from memory, to show the effect of applying continuous pipelining in all cases.

5.3 Evaluation

The enhanced prefetch scheduling techniques we have introduced in this chapter were motivated by problems with the original algorithm as observed in the NAS Parallel benchmarks. We now examine how effectively our new algorithms can address these problems.

When executing these applications on our prototype system, we have encountered several practical limitations that make it difficult for the new scheduling algorithms to achieve their full potential. First, some of the benchmarks are bandwidth-limited, and the hardware we have simply cannot deliver data at the necessary

rate. In these cases, although the new scheduling algorithms smooth the request patterns they cannot substantially decrease the stall time. Second, the algorithms were designed to schedule *prefetches* more effectively, with *releases* being handled by a straightforward application of the original strategy. On IRIX, generating correct release hints is also of critical importance for performance. We have made some minor adjustments to the run-time layer code that handles release requests to better adapt to the new scheduling algorithms, however, this was not the primary focus of this part of the work. As a result, the new scheduling algorithms do a worse job of releasing pages in some cases, masking the benefits of better prefetching.

To address whether the compiler algorithms could be beneficial in a future system capable of delivering higher parallel bandwidth (a network-attached storage system, for instance), and to eliminate the effects of the release hints, we first evaluate the NAS benchmarks on an expanded version of our disk simulator. For these simulations, we provide a more realistic notion of time spent executing a loop, using the number of SUIF instructions in the loop body. Release hints are disabled in the benchmarks. We chose the page size and number of pages to match those on our prototype IRIX system (16kB and 4500 pages, respectively). We charge a modest 100 cycle penalty for issuing a prefetch, which is intended to reflect the IRIX setup where most of the work of a prefetch request is handled by a separate thread. Our simulated disk system continues to have infinite bandwidth.

Table 5.5 gives the overall results of our new algorithms on the NAS Parallel benchmarks with nested loops, obtained via the simulator. For each version, we report the total simulated time, and the simulated stall time, as well as the number of page faults. For the prefetching versions, we report the number of pages that were prefetched (this is the total number of prefetch requests that caused a page to be fetched from disk), the number of prefetched pages that were replaced before they were used (prefetched pages are placed on the top of the LRU stack when the I/O completes), and the number of pages that were not prefetched. The "Not Prefetched" category counts all page faults for which a prefetched page was not pending or in memory, thus some of these pages may have been prefetched, but were replaced before the reference occurred.

From Table 5.5, some broad trends are evident. First, the original pipelining algorithm has vastly more prefetched pages that are replaced before they are used. This reflects the aggressive over-prefetching that we

first discussed with regard to Figure 5.6(e). For MGRID, these extra prefetches are generally beneficial, in that very few page faults are not prefetched, but greater than *fifteen times* the total memory capacity of the simulated system is transferred between disk and main memory unnecessarily. MGRID only pays slightly for this excessive fetching since prefetch requests are cheap, however the additional strain on the memory and I/O bandwidth if these fetches occurred in a real system would be substantial. The excessive prefetches are even worse in APPLU with the original pipelining algorithm. The nested pipelining algorithm has much better memory usage, but this does not translate into an overall reduction in stall time for MGRID, since slightly more pages are not prefetched. For FFTPDE the situation is different. In this benchmark, the replaced pages are not prefetched again before they are needed, and the result is a large number of page faults for the original algorithm. In this case, the nested pipelining algorithm shows a large benefit, both in terms of reduced stall time, and better memory usage.

The continuous pipelining algorithm shows somewhat mixed results. Typically, we see fewer late prefetches than any other algorithm, indicating that our ability to take advantage of multiple loop nests to hide latency is effective. On the other hand, we see more pages replaced before use, and more pages not prefetched as compared to nested pipelining, indicating that we may be prefetching too early in some cases. In two cases (APPBT and APPSP) these effects combine to give continuous pipelining the best overall performance, and in MGRID it is nearly as good as the original algorithm (and better than nested pipelines) with large reductions in unnecessary disk-to-memory traffic. In the other two cases, FFTPDE and APPLU, the continuous pipelining algorithm it not the best performer. For FFTPDE, however, the best choice is nested pipelining, whereas for APPLU the best choice is the original pipelining algorithm. Overall, in the idealized simulator setting, the continuous pipelining algorithm delivers competitive performance for all the benchmarks we examined.

We now consider the effect of our new pipelining algorithms for the NAS Parallel benchmarks with multiply-nested loops on our IRIX prototype system. Figure 5.16 shows the overall execution time of original algorithm using dynamic latency calculations (bars labeled “D”), the nested pipelining algorithm (bars labeled “N”) and the continuous pipelining algorithm (bars labeled “C”), normalized to the original, non-prefetching case (bars labeled “O”). The results are not as dramatic as with the idealized simulator, since our ability to

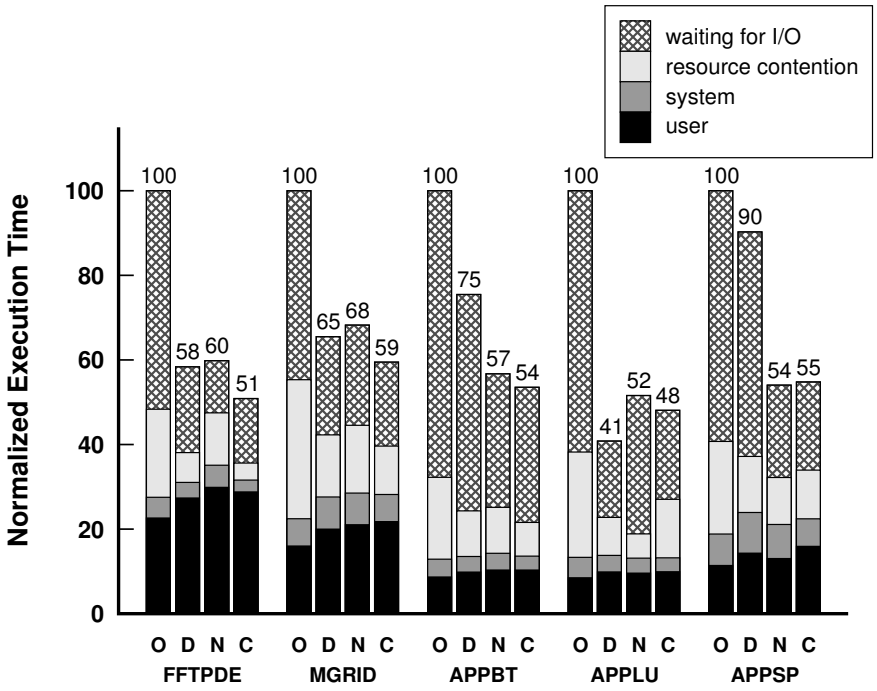


Figure 5.16. Effect of new scheduling algorithms. Bars labeled "O" are the original, non prefetching version; bars labeled "D" use a dynamic latency value with the original scheduling algorithm; bars labeled "N" use the nested pipelining algorithm; bars labeled "C" use the continuous pipelining algorithm.

eliminate stall time is limited by the physical capacity of the machine, however the new algorithms have clear benefits for FFTPDE, APPBT and APPSP.

In FFTPDE and MGRID, nested pipelining produces a small increase in execution time over the original algorithm, primarily the result of an increase in system time. In each of these cases, however, the continuous pipelining algorithm delivers an additional 7% reduction over the original algorithm. For APPBT and APPSP, however, it is the nested pipelining algorithm that makes the largest difference, reducing execution time by 18% and 36% respectively. In these cases, most of the advantage comes from being able to select the best loop nest for software pipelining. A large amount of time is spent in the steady-state once this is done, and so the benefit of avoiding repeated pipeline fills and drains is relatively small. Continuous pipelining has little effect here, leading to an additional 3% reduction in execution time for APPBT and a 1% increase in execution time for APPSP.

In the final case, APPLU, we see that nested pipelining performs worse than the original pipelining algorithm, as was the case for the simulated results. For this benchmark, the continuous pipelining algorithm

results brings the execution time within 7% of the original, reducing the penalty of nested pipelining by nearly half.

Even under the limitations of the real system, we see that the continuous pipelining algorithm has the best, or nearly best, overall performance in 4 of the 5 benchmarks that we studied. In the worst case, we were still able to reduce execution time by 54% relative to the original, non-prefetching case, losing only 6% of the best algorithm's performance. The flexibility of the algorithm, and the ability to adapt to a wide range of conditions, make continuous pipelining a good choice for scheduling prefetches in out-of-core numeric applications.

5.4 Chapter Summary

In this chapter, we demonstrated that generating code to calculate prefetch distances at run-time is a useful extension to the compiler algorithm. It has a negligible performance impact in most cases, and can improve scheduling by itself in some cases by allowing the run-time values of loop bounds to be used, rather than assuming worst-case executions. It also enables more sophisticated techniques for adapting the prefetch distance dynamically, however, an exploration of such techniques is beyond the scope of this thesis.

We also presented and evaluated a progression of refinements to the compiler scheduling algorithm. The final result, continuous software pipelining, allows us to schedule for large latencies in the presence of multiply-nested loops with unknown bounds. It is further capable of handling imperfectly-nested loops correctly. Using our simple disk simulator, we are able to show that the new scheduling algorithm eliminates the bursty behavior of the original algorithm, keeping the pipeline of outstanding prefetches full and increasing opportunities to overlap I/O with computation.

On our prototype IRIX system, the applications that inspired these techniques generally benefited from them, with a large reduction in stall time observed for APPSP, previously the worst benchmark in the suite. In some cases, bandwidth limitations on our prototype system prevented the full potential of the new scheduling algorithms from being visible. However, technology trends such as networked-attached storage suggest that future systems will be able to supply greater bandwidth and should see larger benefits from the continuous

pipelining algorithm. To explore the utility of the algorithms without bandwidth limitations, we used out disk simulator to show that the new algorithms can have even greater benefits for the NAS benchmarks on systems with significantly-higher bandwidth capabilities. Even in cases where there may not be an overall performance benefit, the new algorithms make much more efficient use of memory.

Chapter 6

Conclusions

When will you make an end? — Pope Julius II (from *The Agony and the Ecstasy*)

I/O performance remains a first-order bottleneck for computer systems, as the gap between processor speed and disk latency is only growing wider. For out-of-core applications which rely on disk accesses throughout their execution, techniques that address the I/O bottleneck are essential. As we discussed in Chapter 1, it is important to make good *replacement* decisions to maximize the benefit of caching previously-used pages in memory, as well as to make good *prefetching* decisions to hide the latency of the remaining I/O operations whenever possible. Many techniques have been proposed in the past to address the issue of file system I/O performance, however, we are concerned with improving the performance of a demand-paged virtual memory system. This setting has the potential to simplify the task of writing an out-of-core application by relieving the programmer of concerns related to I/O, provided acceptable performance can be obtained. The use of demand-paged virtual memory, however, also has the potential to degrade the performance of other applications (particularly interactive ones) which must share a machine with an out-of-core program. In this dissertation, we have addressed the open question of whether or not compiler-inserted memory management hints can reduce the execution time of out-of-core applications while limiting their negative impact on interactive applications sharing the same machine, in a demand-paged virtual memory environment.

The key results of this dissertation are as follows:

1. Compiler-inserted *prefetch* and *release* directives can be very effective at reducing the overall execution time of numeric applications. A combination of user-level run-time support and operating system level support is required to adapt the compile-time decisions according to dynamic conditions.

2. The use of replacement hints is surprisingly important on a system with high overhead for making replacement decisions, even if no attempt is made to improve the replacement policy itself. On our IRIX prototype, we found that simulating a hardware reference bit in software led to high contention over locks in the memory system for applications with heavy memory demands. Explicitly identifying pages to be replaced reduced the need for the operating system to collect detailed page usage information, resulting in a substantial decrease in lock contention and a corresponding reduction in execution time. Although the goal of this dissertation is not to investigate hardware support, this result argues strongly for the inclusion of a reference bit in the hardware memory management unit.

3. Allowing memory-intensive applications more control over their paging strategies can alleviate the pressure on other applications, leading to vastly improved responsiveness for interactive jobs.

4. Adaptation to run-time conditions is critical when dealing with the large and variable latencies that occur with disk I/O. Compiler-generated code should not require a fixed pre-specified latency parameter to schedule prefetch operations. We demonstrated that calculating prefetch distances at run-time does not have a negative impact on performance, and in one case resulted in a substantial improvement. We also introduced new software pipelining algorithms capable of handling very large latencies in the presence of multiply-nested loops with unknown bounds. Although the benchmarks that inspired these techniques remain inherently I/O bound, we showed that future systems with greater bandwidth capabilities will be able to benefit from them.

6.1 Future Work

The ultimate goal of our research on I/O prefetching for out-of-core applications is to reduce the I/O bottleneck as much as possible, expanding the range of systems on which it is feasible to conduct large-scale

scientific computing research. To this end, we desire the ability to run unmodified applications on either a high-end supercomputer with massive physical memory capacity, or a more modest small-scale server system. We consider recompilation for a range of systems to be a reasonable requirement, while manually re-writing the code to deal with I/O requirements on a smaller system is too large a burden. In this section, we briefly discuss how our research can be extended to further improve the performance of out-of-core applications.

In Section 5.1.1 we described several options for determining an appropriate prefetch distance at run-time. An obvious next step is to explore these alternatives. Although setting the prefetch distance at the entry to a set of loops is straightforward, it may be necessary to investigate strategies for adjusting the prefetch distance as the loops execute.

We observed that several of our benchmarks remain I/O-bound, despite our best efforts to schedule the prefetches more effectively. In cases where I/O bandwidth is the limiting factor, performance can be improved with better layouts of pages to swap disks. In the context of parallel file systems, the need to match the layout of data on disk to application access patterns has been well-studied [17, 33, 62]. For swap I/O, however, the layout is largely controlled by the operating system pager and is optimized for generating large numbers of clean pages quickly, without regard for the application's access pattern. The use of replacement hints to manage the layout of pages on swap space deserves further study. Strategies such as striping, replication, and extents that have been deployed in persistent file systems should be investigated for swap space. We anticipate that the compiler analysis will need to be extended to a larger program scope for these decisions, since the best layout on swap is determined by the shape of the next access to the data, which may occur in a different procedure entirely.

Finally, the new software pipelining algorithms introduced in this dissertation are designed to address the problem of hiding long latencies. Similar problems are likely to occur for applications that use network I/O, such as grid computing problems. Exploring compiler-directed prefetching using the new scheduling algorithms in these settings is another direction for future work.

6.2 Final Observations

The research behind this dissertation has led to several insights and lessons about working with computer systems that are not specific to the problem of prefetching page faults in virtual memory systems. These larger lessons may be obvious in hindsight, but they may be of use to others doing research in the broader systems area. We thus discuss three of them here.

- *Be careful of the effects of assuming that a particular phenomenon will always be a performance bottleneck.* For example, in IRIX (as in many operating systems), there is an underlying assumption that once an application begins paging, performance will suffer so severely that there is little point optimizing software operations. Other performance killers are hidden by the magnitude of the cost of I/O. Simulating reference bits in software, and locking disciplines that prevent new memory from being referenced while pages are being written out are examples of choices that are only reasonable if paging activity represents a severe loss of performance anyway. Eliminating the ”smoking gun” performance problem (stalls due to page faults in this example) may not immediately lead to better performance because all it does is expose another limitation that was previously masked.

- *Good performance monitoring tools are critical for developing complex high-performance software.* Without a reliable means to attribute execution time (or other metrics of interest) to particular events, debugging a performance problem can be virtually impossible.

- *Cooperation between different levels and components of a complex system is more effective than having each component make a best guess about what needs to be done.* In this dissertation, we have explored cooperation between compile-time and run-time, and between user-level and operating system code to enable better memory management for some applications. We had an explicit goal of requiring no intervention on the part of the application programmer. In the real world, however, including the programmer as a “cooperative component” that can provide hints to the rest of the system is also an effective approach. For example, in our implementation we compile the original program to C source code that includes the prefetch and release directives. In combination with the performance monitoring

and reporting infrastructure that we developed, this makes it tractable for the compiler to handle most of the references, and the programmer to manually insert prefetches for a small number of key references that the compiler was unable to analyze completely. We have observed several cases of this sort, where the surrounding code has already been transformed into an appropriately-pipelined form, so the manual effort is simply a matter of copying a prefetch call and replacing the memory address. The additional compiler effort to capture these corner cases is probably not justified and it is valuable to provide those programmers that have extreme performance requirements with the tools to understand and modify the results of the automatic transformations.

Bibliography

[1] Tom Vander Aa, Francisco Barat, Murali Jayapala, Henk Corporaal, Francky Catthoor, and Geert Deconinck. Software transformations to reduce instruction memory power consumption using a loop buffer. In *Proceedings of the 1st Workshop on Optimizations for DSP and Embedded Systems (ODES)*, San Francisco, CA, USA, March 2003.

[2] Randy Allen, Ken Kennedy, and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[3] Meenakshi Arunachalam, Alok Choudhary, and Brad 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, Ottawa, Ontario, Canada, May 1995. Extended Abstract.

[4] Ozalp Babaoglu and William Joy. Converting a swap-based system to do paging in an architecture lacking page-referenced bits. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 78–86, Pacific Grove, California, USA, December 1981.

[5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[6] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel

Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[7] Gretta Bartels, Anna Karlin, Darrell Anderson, Jeffrey Chase, Henry Levy, and Geoffrey Voelker. Potentials and limitations of fault-based Markov prefetching for virtual memory pages. In *Proceedings of the 1999 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 206–207, Atlanta, Georgia, USA, May 1999. Poster.

[8] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain, Colorado, USA, December 1995.

[9] Rajesh Bordawekar, Alok Choudhary, and J. Ramanujam. Automatic optimization of communication in compiling out-of-core stencil codes. In *Proceedings of the 10th ACM International Conference on Supercomputing*, Philadelphia, Pennsylvania, USA, May 1996.

[10] Angela Demke Brown and Todd C. Mowry. Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 31–44, San Diego, California, USA, October 2000.

[11] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 188–197, Ottawa, Ontario, Canada, May 1995.

[12] Fay Chang and Garth A. Gibson. Automatic I/O Hint Generation through Speculative Execution. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, USA, February 1999.

[13] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[14] Wesley W. Chu and Holger Opderbeck. The page fault frequency replacement algorithm. In *AFIPS Conference Proceedings, Fall Joint Computer Conference*, volume 41, pages 597–609. AFIPS Press, 1972.

[15] Fernando J. Corbato. A paging experiment with the multics system. In Herman Feshbach and K. Uno Ingard, editors, *In Honor of Philip M. Morse*, pages 217–228. MIT Press, Cambridge, MA, 1969.

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

[17] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, New Hampshire, USA, 1993.

[18] Thomas M. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[19] Michael Cox and David Ellsworth. Application-Controlled Demand Paging for Out-of-Core Visualization. In *Proceedings of the 8th conference on Visualization '97*, Phoenix, Arizona, USA, October 1997.

[20] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In *Proceedings of the 1993 ACM SIGMOD Conference on Management of Data*, pages 257–266, Washington, D.C., USA, May 1993.

[21] Angela K. Demke. Automatic I/O Prefetching for Out-of-Core Applications. Master's thesis, University of Toronto, Department of Computer Science, January 1997.

[22] John Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Communications of the ACM*, 4(10):435–436, October 1961.

[23] Benjamin Gamsa, Orran Krieger, and Michael Stumm. Optimizing IPC performance for shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 208–211, Boca Raton, Florida, USA, August 1994.

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

[25] Andrew S. Grimshaw and Edmond C. Loyot, Jr. ELFS: object-oriented extensible file systems. Technical Report TR-91-14, Department of Computer Science, University of Virgina, July 1991.

[26] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[27] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197, Boston, Massachusetts, USA, October 1992.

[28] James V. Huber, Jr., Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, Spain, July 1995.

[29] IEEE. Threads Extension for Portable Operating Systems (Draft 7), February 1992.

[30] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth MacKenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, Saint Malo, France, October 1997.

[31] Ken Kennedy, Charles Koelbel, and Michael 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.

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

[33] Orran Krieger and Michael Stumm. HFS: A performance-oriented flexible file system based on building-block compositions. *ACM Transactions on Computer Systems*, 15(3):286–321, August 1997.

[34] Thomas M. Kroeger and Darrell D. E. Long. Predicting file system actions from prior events. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 319–328, San Diego, California, USA, January 1996.

[35] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, Georgia, USA, June 1988.

[36] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, Denver, Colorado, USA, June 1997.

[37] Nathaniel McIntosh. *Compiler Support for Software Prefetching*. PhD thesis, Rice University, May 1998. Technical Report TR98-303.

[38] Dylan McNamee and Katherine Armstrong. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Proceedings of the USENIX Association Mach Workshop*, pages 17–29, Burlington, Vermont, USA, October 1990.

[39] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994. Technical Report CSL-TR-94-626.

[40] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 3–17, Seattle, Washington, USA, October 1996.

[41] Todd C. Mowry, Monica S. Lam, and Anoop 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*, pages 62–73, Boston, Massachusetts, USA, October 1992.

[42] Michael Paleczny, Ken Kennedy, and Charles 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, Virginia, USA, February 1995.

[43] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 79–95, Copper Mountain, Colorado, USA, December 1995.

[44] Richard Rashid, Avadis Tevanian, Jr., Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, Palo Alto, California, United States, October 1987.

[45] Seagate Technology, Inc. *Cheetah 4LP Family: ST34501N/W/WC/WD/DC, Product Manual, Volume 1*, July 1997. Publication number: 83329120, Rev. B.

[46] Silicon Graphics, Inc. *SGI Irix 6.5* `aio_read` *manual page*.

[47] Christopher Small and Margo Seltzer. A Comparison of OS Extension Technologies. In *Proceedings of the 1996 Usenix Technical Conference*, San Diego, California, USA, January 1996.

[48] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(12):7–21, December 1978.

[49] Inshik Song and Yookun Cho. Page prefetching based on fault history. In *USENIX Mach III symposium proceedings*, pages 203–213, Sante Fe, New Mexico, USA, April 1993.

[50] Evan Speight and Martin Burtscher. Delphi: Prediction-based page prefetching to improve the performance of shared virtual memory systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 49–55, Las Vegas, Nevada, USA, June 2002.

[51] Indira Subramanian. Managing Discardable Pages with an External Pager. In *Proceedings of the 2nd USENIX Mach Symposium*, pages 77–86, Monterey, California, USA, November 1991.

[52] Sun Microsystems. *SunOS 5.9* `aio_read` *manual page*.

[53] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, , and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 1–14, San Diego, CA, USA, January 1996. Usenix.

[54] Rajeev Thakur, Rajesh Bordawekar, and Alok 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.

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

[56] Ron Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9(1/2):105–134, 1995.

[57] U.S. Department of Energy, Office of Advanced Scientific Computing Research and Office of Biological and Environmental Research. Report on the Computational Infrastructure Workshop for the Genomes to Life Program, March 2002.

[58] Vivekanand Vellanki and Ann Chervenak. A cost-benefit scheme for high performance predictive prefetching. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, Portland, Oregon, USA, November 1999.

[59] Geoffrey M. Voelker, Eric J. Anderson, Tracy Kimbrel, Michael J. Feeley, Jeffrey S. Chase, Anna R. Karlin, and Henry M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 33–43, Madison, Wisconsin, USA, June 1998. ACM Press.

[60] Zvonko G. Vranesic, Michael Stumm, David M. Lewis, and Ron White. Hector: A hierarchically structured shared-memory multiprocessor. *IEEE Computer*, 24(1):72–79, January 1991.

[61] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12), December 1994.

[62] David Womble, David Greenberg, Stephen Wheat, and Rolf Riesen. Beyond core: Making parallel computer I/O practical. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 56–63, Hanover, New Hampshire, USA, 1993.

[63] Yuanyuan Zhou, Limin Wang, Douglas W. Clark, and Kai Li. Thread scheduling for out-of-core applications with memory server on multicomputers. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 57–67, Atlanta, Georgia, USA, May 1999. ACM Press.