Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently

Angela Demke Brown and Todd C. Mowry
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{demke,tcm}@cs.cmu.edu

Abstract

Out-of-core applications consume physical resources at a rapid rate, causing interactive applications sharing the same machine to exhibit poor response times. This behavior is the result of default resource management strategies in the OS that are inappropriate for memory-intensive applications. Using an approach that integrates compiler analysis with simple OS support and a run-time layer that adapts to dynamic conditions, we have shown that the impact of out-of-core applications on interactive ones can be greatly mitigated. A combination of prefetching *pages that will soon be needed, and* releasing *pages no longer in use results in good throughput for the out-of-core task and good response time for the interactive one. Each class of application performs well according to the metric most important to it. In addition, the OS does not need to attempt to identify these application classes, or modify its default resource management policies in any way. We also observe that when an out-of-core application releases pages, it both improves the response time of interactive tasks, and also improves its own performance through better replacement decisions and reduced memory management overhead.*

1 Introduction

Many of the computational problems of interest to scientists and engineers involve data sets that are much larger than physical memory [6, 7, 17]. Despite the continuing trend toward larger memories, it is unlikely that these data sets will ever fit entirely within main memory. Increases in processor power and memory capacity make it feasible to solve larger problems, or to solve the same problem at a finer granularity, but the size of the data set grows with the problem being solved. For instance, input data sets for scientific visualization can currently exceed 100 Gbytes [5]. For these "out-of-core" applications, I/O is required throughout the execution of the program to bring data into memory as it is needed and possibly to move it back out to disk. Performance concerns have traditionally forced programmers to explicitly manage the I/O in their out-of-core codes. Recently, however, we demonstrated that paged virtual memory can be enhanced with prefetching to effectively hide the latency of page faults without placing any burden on the programmer [15]. In this approach, the compiler provides information on future access patterns, the OS supports a simple *prefetch/release* interface, and a run-time layer improves performance by adapting to dynamic behavior.

While this earlier work demonstrated that out-of-core applications can achieve excellent performance on a dedicated machine, 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.

1.1 Impact on Interactive Performance

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 OS. While 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. 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 remain responsive regardless of the intervening delay.

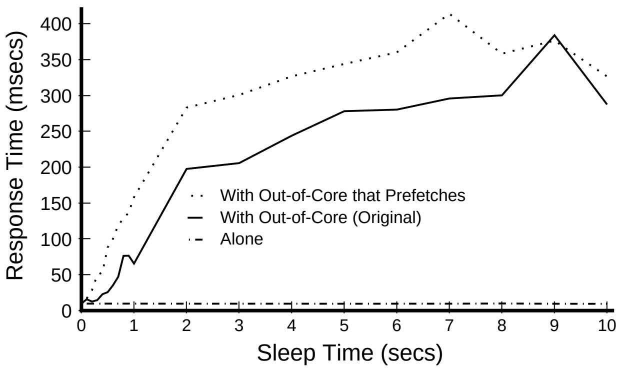


Figure 1. Impact of sharing the machine with an out-of-core matrix-vector multiplication (MATVEC) on the response time of an interactive task across a range of sleep times between touching 1 MB of data.

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.[1] 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 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 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 OS policies, a significant amount of recent research has focused on customizable operating systems. While a customizable OS could provide the flexibility to tailor the resource management policies for out-of-core codes, our results in this paper 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. To accomplish this goal, we adopt a strategy similar to our earlier work [15] in which the OS, compiler, and a run-time layer all cooperate. The role of the OS is to perform global resource allocation 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.

[1] This amount of memory is artificially low for modern systems, but makes it possible to run experiments on out-of-core programs in a reasonable amount of time. Similar behavior can be seen with more memory and larger out-of-core programs, although the time required to consume all physical memory increases with the amount of memory available

1.2 Objectives of This Study

In our earlier study [15], our focus was using *prefetching* to hide the *I/O latency* of out-of-core applications running on a *dedicated* machine. In this study, we focus on using *release* operations to *manage physical memory* intelligently within a *multiprogramming workload* that includes an out-of-core application. Although we introduced the concept of release operations in that earlier paper, we made little use of them because they offered no significant performance benefit to stand-alone out-of-core applications on the research prototype OS (Hurricane [21]) and machine (Hector [22]) that we used. Note that we observe a different result in this study using a modern commercial OS and machine.

The primary contribution of this paper is that we propose, implement, and evaluate a solution to the problem of preventing out-of-core applications from ruining the response time of interactive applications while still enjoying the performance benefits of aggressive I/O prefetching. Our solution uses the compiler to automatically insert *release* hints (in addition to *prefetch* hints) into the out-of-core application while a run-time layer and OS provide appropriate support. This approach requires minimal changes to existing operating systems and places no additional burden on the programmer. We implement our solution within a modern commercial system (an SGI Origin 200 running our modified version of IRIX 6.5) and evaluate its performance impact on both out-of-core applications and interactive tasks sharing the same machine.

The remainder of this paper is organized as follows. Section 2 motivates allowing applications to manage their own resources, and describes the features we feel are needed to do so effectively. Section 3 describes the components of our system and their implementations. Section 4 presents our experimental results, and we discuss related work and draw conclusions in Sections 5 and 6.

2 Memory Management Strategies

The goal of a virtual memory management system in a multiprocessor 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. Finally, we outline the features we believe are necessary for an effective system that allows applications to explicitly manage their memory resources.

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 OS. In practice, most workstation operating systems use global page replacement.

Although local replacement policies can 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 OS, 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 OS. The largest drawback of specializing applications to do memory management is the burden placed on the programmer; however, we propose a framework in which all the necessary modifications are performed automatically by a compiler.

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 OS supports external pagers to allow applications to control the backing storage of their memory objects [18]. Extensions to the external pager interface have been used to implement user-level page replacement polices [14], and to support discardable pages (i.e. dirty pages that do not need to be written to backing store) [20]. 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 performance since the gains they enjoy impose a heavy penalty on other processes sharing the system. In this case, the OS could require that prefetching applications also explicitly release pages.

Given that application-controlled memory management is desirable, one possibility is for the OS 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 OS 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 OS 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 [19], for example. A reactive system benefits applications that can make better replacement decisions than the default OS 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 OS 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 OS 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 goal is to develop a system that allows applications to pro-actively return memory to the system on a page-by-page basis, to the mutual benefit of themselves and other concurrently executing applications without placing any additional burden on the programmer. We now outline the elements that we believe are necessary to achieve this goal.

2.3 Requirements for Effective Application-Directed Memory Management

If applications are to manage their own memory usage, the first requirement is some form of support from the OS for this type of activity. Second, to automate memory management without rewriting the application source code, we will need compiler analysis to detect access patterns and insert the necessary paging operations. Finally, since good replacement decisions will depend on dynamic conditions during program execution, we will need a run-time layer to intercept the information provided by the compiler and adapt the application's behavior as required.

2.3.1 Operating System Support

The OS must supply both primitive operations and additional information to applications. The operations should allow the application to specify the virtual memory addresses that it will need in the future as well as those that it no longer needs. The additional information is needed to allow the application to make informed decisions about when memory management activity is required. It should include information about which virtual pages are currently in memory, how many pages are currently in use, and the upper limit on pages that the application should use.

2.3.2 Compiler and Run-time 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

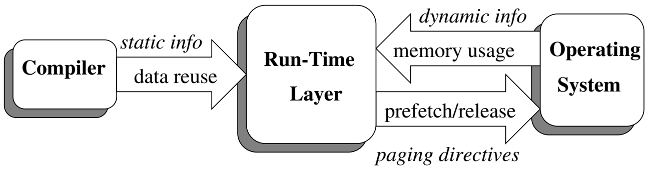


Figure 2. Information flow between components of our system.

in the future? If not, then a release hint is inserted. Second, does the number of other unique pages that will be accessed before the page is reused exceed the expected amount of available memory? If so, then the page is unlikely to remain in memory, and a release hint is inserted. Otherwise, release hints are not inserted.

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* reuse (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 [8, 15], 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 compiler is that it decides when reuse can be exploited 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, a 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. The role of the run-time layer is to use the information provided by the OS and the compiler to answer the following questions: When should memory be returned to the OS? How many pages should be released? Which of the "releasable" pages should actually be given up? Figure 2 depicts the flow of information from the compiler and the OS to the run-time layer.

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 OS 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 OS. 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.

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 OS, thus keeping at least the first portion of the array in memory for future use.

2.4 An Example

To help illustrate these concepts, we now present a simple example. Figure 3(a) shows the source code for a calculation that averages an element of a matrix with its neighbors, while Figure 3(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

(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

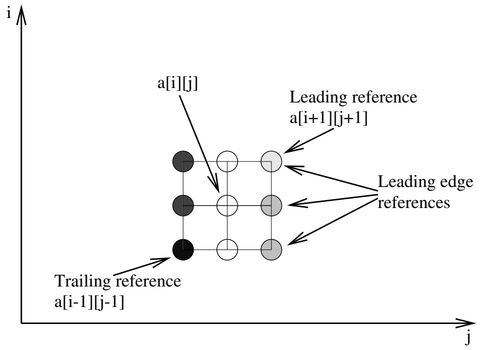


Figure 3. Example source code showing multiple references with different types of reuse, and graphical view of the data accesses during a single iteration of the innermost loop.

(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.

Having outlined the features that we believe are necessary to achieve a good pro-active user-level memory management system, we turn now to a discussion of the specific components in our prototype system.

3 Overview of Prototype System

Our prototype system consists of three major components: extensions to the OS, a compiler analysis pass, and a run-time layer. We now describe these components.

3.1 Implementation of OS Support

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 *policy modules* (PMs) that allow users to select various memory management policies for page size, allocation, migration, and replication. A PM 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 PM—called "PagingDirected"—that allows a user-level process to invoke prefetch and release operations on pages of its address space. In addition, the PagingDirected PM shares information about memory usage with the application through a single 16KB page.

3.1.1 Managing the Shared Page

The shared page is allocated by the OS and mapped read-only into the application's address space when the PagingDirected PM is created. The page is used primarily as a bitmap, indexed by virtual page number, in which bits are set to indicate that the corresponding page is in memory, and cleared otherwise. The first two words in the page are reserved, however, to indicate the current number of pages in use by the process, and the upper limit on pages that the process should be using, respectively.

All updates to the shared page are handled by the OS. When the PagingDirected PM is created, all bits in the shared page are initially set. When the application attaches the PM to a region of its virtual address space, the bits corresponding to those addresses are all cleared. Thereafter, bits are set whenever a physical page is allocated for a virtual page associated with this PM, 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. 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 OS 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. Another alternative that we have not explored would be to notify interested applications if conditions change by more than a set threshold, rather than waiting for memory activity to occur.

3.1.2 Handling Prefetch and Release Requests

When the PagingDirected PM 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, the request is discarded immediately. This feature prevents memory from being stolen to satisfy prefetches when the demand for memory is high. Second, when the request completes, the prefetched page is not fully validated and no entry is made in the TLB. This feature prevents mappings for prefetched pages from displacing TLB entries which are still in use.

Requests to release pages are handled by passing the addresses to a new system releasing daemon—called the *releaser*—which functions similarly to the paging daemon, but is specialized to reclaim only the pages indicated by the application. When a release request is made, the PagingDirected PM clears the bits for the pages and enters the request in the releaser's work queue. The releaser handles requests 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) since the time of the request. The releaser then performs all actions needed to free the pages, including writing back dirty pages. Released pages are placed at the end of the free list, giving pages that were released too early a chance to be rescued.

3.1.3 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 PM 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:

$$\begin{aligned} \textit{upper limit} \quad &= \quad \min(\textit{max\_rss}, (\textit{current\_size} + \\ &\qquad \textit{tot\_freemem} - \textit{min\_freemem})) \end{aligned} \quad (1)$$

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. Finding the right amount of memory for each process is beyond the scope of this paper.

3.2 Implementation of Compiler Analysis

We implemented our compiler algorithm as a pass in the SUIF (Stanford University Intermediate Format) compiler [9]. This algorithm is an extension of the algorithm we developed earlier for inserting prefetching hints into array-based codes [15]; pointer-based data structures are not currently handled, although techniques used for cache prefetching may be applicable [13]. We now briefly describe our algorithm. The following parameters are given to the compiler to describe the target system: the size of main memory, the page size, and the page fault latency. The compiler first uses *reuse analysis* to detect the intrinsic data reuses in the access patterns, then uses the page size and memory size parameters to apply *locality analysis* to predict when misses (i.e. page faults) are likely to occur. References that are likely to suffer page faults are isolated through *loop splitting* techniques, and prefetches for these references are scheduled based on the latency parameter using *software pipelining*. Figure 4 shows the process of creating the specialized executable from the original source code. The compiler analyzes each set of nested loops independently, thus reuses that occur between independent sets of loops are not considered. While the earlier algorithm did insert release hints in some cases, we have extended that analysis in two major ways: (i) we insert releases far more aggressively, and (ii) we encode reuse information into the release hints to allow the runtime layer to choose which pages to release first.

Given the existing locality analysis, it is relatively straightforward to generate release operations. During locality analysis, the compiler identifies groups of references that effectively share the same data and can be treated as a single reference—this is called "group locality". For each of these groups (a group may contain only a single reference), the compiler identifies the *leading reference*

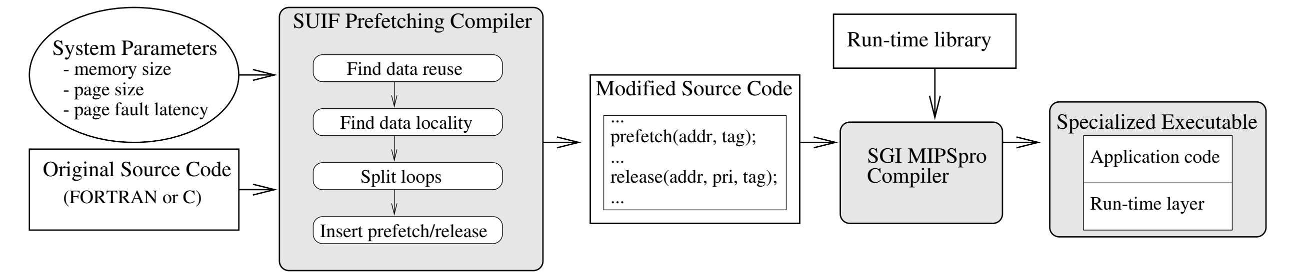


Figure 4. Steps in the automatic transformation of original application into prefetching/releasing executable.

(i.e. the first reference to access the data) as the reference to prefetch—we simply extend this analysis to also identify the *trailing reference* (the last one to touch the data) as the address to release. 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:

$$\mathit{priority}(x) \;=\; \sum_{i\,\in\,\mathit{temporal}(x)} 2^{\,\mathit{depth}(i)} \qquad (2)$$

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 5 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 (2), 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 since this item is smaller than a page and is expected to remain in memory.

(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];
```

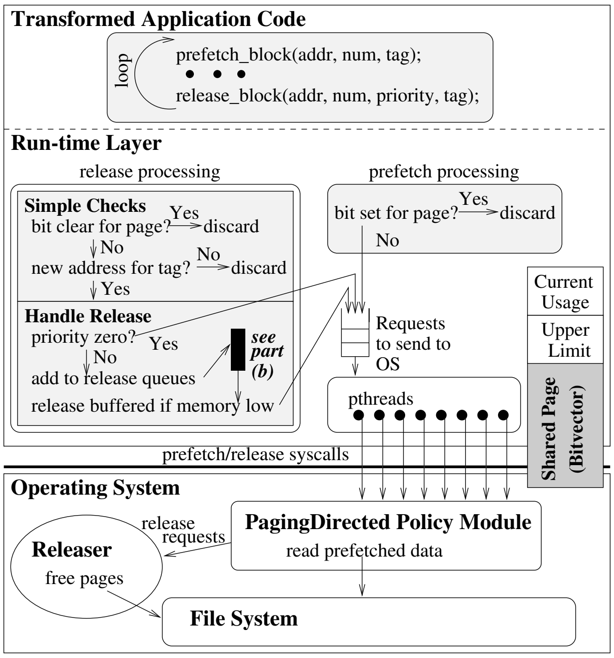
(b) Code with Prefetch and Release

```
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);
  }
}
```

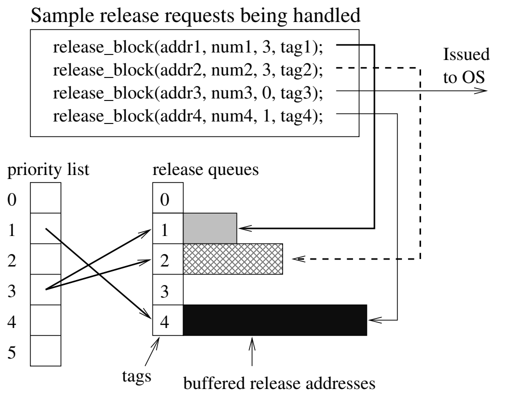
Figure 5. Example of the output of the prefetching compiler. Arguments are: (prefetch address, release address, number of 16KB pages, release priority, request identifier)

3.3 Implementation of the Run-time Layer

Figure 6 illustrates how prefetches and releases are processed by the run-time layer. 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 disk subsystem. The run-time layer accomplishes these requirements by creating a number of *pthreads* [11] that make the actual calls to the PagingDirected PM 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 checked to see if a prefetch is really needed. If so, 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



(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 6. Handling prefetches and releases at run-time.

the PagingDirected PM. We chose to use a pthreads-based approach since the IRIX kernel does not provide asynchronous I/O to user-level programs. Rather than attempt to add this functionality to IRIX, we chose an approach very similar to the implementation of the asynchronous I/O library in IRIX.

The same set of pthreads are also used to actually issue the release requests to the OS. 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 OS at the time when they are encountered, while the other buffers releases based on the compiler-inserted priorities and only issues requests when necessary, based on the information provided by the OS. 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 OS. 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 6(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. (We have not experimented with varying this parameter.)

As we will show in Section 4, 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.

4 Experimental Results

To evaluate the concepts presented in this paper, we ran several out-of-core applications with the simulated interactive task described in Section 1.1. We will first describe the platform used to obtain these results, then look at the impact of prefetching, alone and with both aggressive releasing and release buffering, on the execution time of the

Table 1. Experimental platform characteristics.

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
Track-to-track seek, read:	18 msec (typical)
Track-to-track seek, write:	19 msec(typical)
Number of SCSI controllers:	5
Disks per controller:	2

Table 2. 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	452 MB (600%)	23.9
MATVEC	matrix-vector multiply	10^2 x 10^6 matrix, 10^6 vector	404 MB (539%)	11.1

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.

4.1 Hardware Platform

Our experimental results were obtained on a 4-processor SGI Origin 200, running our modified version of the IRIX 6.5 operating system. 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 buses on the Origin. The basic hardware characteristics of our system are summarized in Table 1.

4.2 Benchmarks

We performed our experiments using out-of-core versions of five applications taken from the NAS Parallel benchmark suite [1] as well as a matrix-vector multiplication kernel (MATVEC). The code for MATVEC was shown earlier in Figure 5(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 2 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. We believe that any additional improvements to the results shown here will come from improved compiler analysis and code generation, and greater run-time layer involvement, rather than from additional operating system support.

4.3 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 7, we show the execution times of the out-of-core programs, normalized to the original case. 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

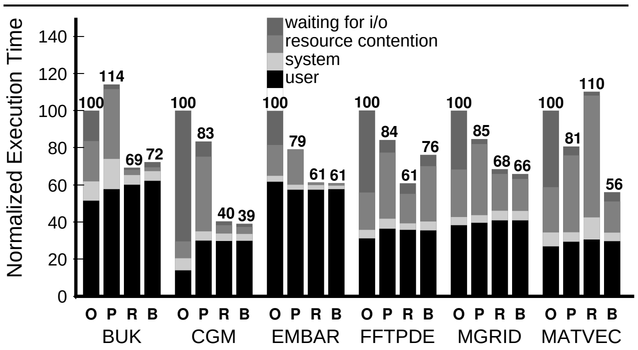


Figure 7. Impact of prefetching and releasing on the execution times of the out-of-core applications. (O = original, P = with prefetching, R = with prefetching and releasing, B = with prefetching and release buffering)

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. Because we use separate threads to issue the prefetch requests, the prefetch service time does not appear in the execution time of the main application. Since we are using a multiprocessor, many of the prefetches can be serviced in parallel. Although the prefetch threads compete with the main application and the interactive task for CPU time, it is a very small effect since these threads spend most of their time waiting for I/O.

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. 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.

The I/O stall reductions, and the system time and user time components of these experiments all validate the results we obtained in our previous study on compiler-based I/O prefetching [15], demonstrating that these techniques are still applicable with modern hardware and software. Our prior study, however, showed that releasing memory provided no significant benefit to the out-of-core applications over prefetching alone. One key difference here is that the earlier compiler implementation did not insert release operations in many situations. 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 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 7, 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

Table 3. Pages freed by system or by release, and pages rescued from the free list.

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

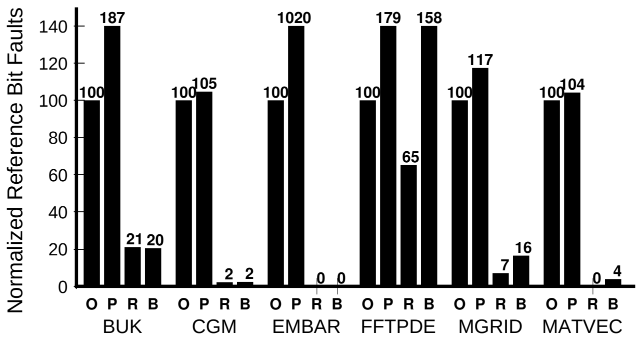


Figure 8. Soft page faults due to page invalidations.

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 8 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.

4.4 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 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 3 for the original out-of-core programs and the versions that both prefetch and release memory without buffering.

From Table 3, 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 been re-allocated to hold other data before being referenced again, and the reused

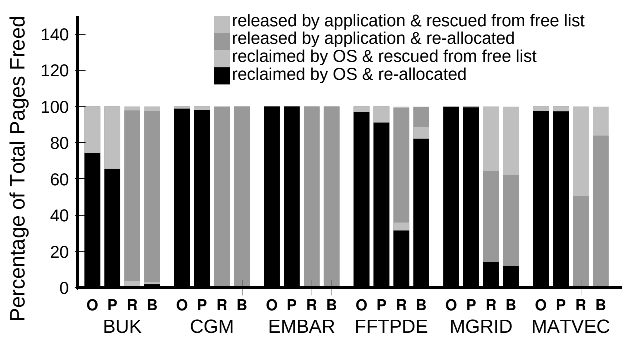
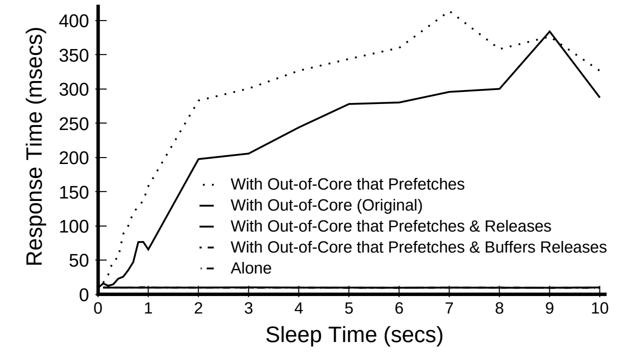
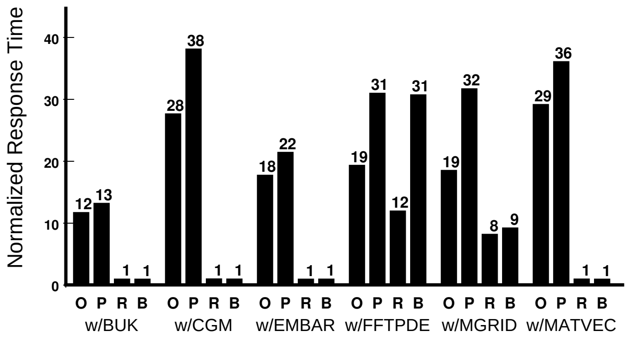


Figure 9. Breakdown of outcomes for freed pages.



(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 10. Impact of releasing on interactive response time.

data will need to be brought back into memory from swap.

Figure 9 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 9, 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 reallocated 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 3, 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.

4.5 Impact on Interactive Response Time

Figure 10 gives an overview of the performance improvements obtained for the "interactive" task. In Figure 10(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 1.1, 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 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 10(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 10(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 10(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.

5 Related Work

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 OS 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 [18]. Extensions to the external pager interface have been used to implement user-level page replacement policies [14] and to support discardable pages (i.e. dirty pages that do not have to be written to backing store) [20]. More aggressive application control of physical memory was implemented in the V++ kernel by Harty and Cheriton [10]. 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 [12], SPIN [2], and Vino [19]. 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.

Other related work has shown the importance of considering both prefetching and replacement decisions in tandem, in the context of I/O prefetching for file system references. Cao *et al.* [3] 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.* [16] 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. In the original TIP implementation, applications had to be manually modified to generate the necessary access hints. Recently, another approach for automatically modifying applications to provide hints about their future accesses has been presented by Chang and Gibson [4]. Applications are modified automatically (using a binary modification tool on the program executable) to speculatively execute the code and generate access pattern hints to be passed to the TIP system. Because it is much more costly to track all virtual memory references (versus explicit file requests only) the techniques used by the TIP system for deciding what to eject from the file cache are not especially applicable for virtual memory management.

6 Conclusions

We have implemented and evaluated a complete and fully-automatic system which exploits compiler-inserted *release* operations to intelligently manage the physical memory resources of out-of-core applications. These specialized applications can reduce their impact on the performance of other applications while still exploiting aggressive prefetching to hide their I/O latency. Our results confirm that compiler-inserted I/O prefetching works well on commercial operating systems and state-of-the-art machines (even though faster processors make it much more challenging to hide the I/O latency), hiding roughly 85-100% of the I/O stall time in our out-of-core benchmarks and achieving good overall speedups.

The significant benefit to the out-of-core benchmarks due to aggressively releasing memory was mostly unexpected. In BUK we expected to see a benefit from improving on the replacement policy, but for the other applications (excepting MATVEC, which is hurt by aggressive releasing), the improvement comes from reducing the interference between the operating system and the application. We found the extent of this interference between the paging daemon and the page fault handling to be especially surprising. Not only does the paging daemon greatly increase the number of soft page faults as it attempts to simulate reference bits in software, but the time to handle these page faults is also inflated by increased lock contention. Because the overhead of determining which pages to replace is so large, explicit replacement hints can improve performance, even if they are not making better replacement decisions than the default policy. It would be interesting to see if these benefits still occur on a system with hardware reference bits (although such a study was beyond the scope of this paper since IRIX only runs on MIPS processors).

Overall, our compiler-based approach for combining both prefetching and releasing to allow out-of-core applications to explicitly manage their virtual memory is a situation in which everyone wins. Both the memory-intensive programs and the less demanding interactive ones sharing the system obtain performance benefits. Only the out-of-core programs need to be modified, and the changes are performed automatically by the compiler without burdening the application programmer. Furthermore, the default policies of the operating system do not need to be changed, and no overhead is introduced in the common case for managing ordinary applications.

7 Acknowledgements

We thank Andrew Myers (our shepherd) for helping us improve the presentation of this paper. This research is supported by a grant from NASA. Todd C. Mowry is partially supported by an Alfred P. Sloan Research Fellowship.

References

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

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. of the 15th Symp. on Operating System Principles*, Dec. 1995.

[3] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 188–197, 1995.

[4] F. Chang and G. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proc. of the 3rd OSDI*, Feb. 1999.

[5] M. Cox and D. Ellsworth. Application-Controlled Demand Paging for Out-of-Core Visualization. In *Proc. of Visualization '97*, Oct. 1997.

[6] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/Output Characteristics of Scalable Parallel Applications. In *Proc. of Supercomputing '95*, Dec. 1995.

[7] J. M. del Rosario and A. Choudhary. High Performance I/O for Massively Parallel Computers: Problems and Prospects. *IEEE Computer*, 27(3):59–68, Mar. 1994.

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

[9] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.

[10] K. Harty and D. Cheriton. Application-Controlled Physical Memory Using External Page-Cache Management. In *Proc. of the 5th ASPLOS*, pages 187–199, Oct. 1992.

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

[12] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceo, R. Hunt, D. Mazires, T. Pinckney, R. Grimm, J. Jannotti, and K. MacKenzie. Application Performance and Flexibility on Exokernel Systems. In *Proc. of the 16th Symp. on Operating System Principles*, Oct. 1997.

[13] C.-K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proc. of the 7th ASPLOS*, pages 222–233, Oct. 1996.

[14] D. McNamee and K. Armstrong. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Proc. of the USENIX Assoc. Mach Workshop*, pages 17–29, 1990.

[15] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *Proc. of the 2nd OSDI*, pages 3–17, Oct. 1996.

[16] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proc. of the 15th Symp. on Operating System Principles*, pages 79–95, Dec. 1995.

[17] J. T. Poole. Preliminary Survey of I/O Intensive Applications. Technical Report CCSF-38, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[18] R. Rashid, A. Tevanian, Jr., M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proc. of the 2nd ASPLOS*, Oct. 1987.

[19] C. Small and M. Seltzer. A Comparison of OS Extension Technologies. In *Proc. of the 1996 USENIX Technical Conference*, Jan. 1996.

[20] I. Subramanian. Managing Discardable Pages with an External Pager. In *Proc. of the USENIX Mach Symposium*, Nov. 1991.

[21] R. C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design. *Journal of Supercomputing*, 9(1/2):105–134, 1995.

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