The Compression Cache: Using On-line Compression to Extend Physical Memory[*]

Fred Douglis
douglis@research.panasonic.com
Matsushita Information Technology Laboratory
Two Research Way
Princeton, NJ 08540-6628 USA

January, 1993

Abstract

This paper describes a method for trading off computation for disk or network I/O by using less expensive on-line compression. By using some memory to store data in compressed format, it may be possible to fit the working set of one or more large applications in relatively small memory. For working sets that are too large to fit in memory even when compressed, compression still provides a benefit by reducing bandwidth and space requirements.

Overall, the effectiveness of this *compression cache* depends on application behavior and the relative costs of compression and I/O. Measurements using Sprite on a DECstation[1] 5000/200 workstation with a local disk indicate that some memory-intensive applications running with a compression cache can run two to three times faster than on an unmodified system. Better speedups would be expected in a system with a greater disparity between the speed of its processor and the bandwidth to its backing store.

1 Introduction

Over the past decade, the processing power and physical memory size of typical computers have increased dramatically. Even as workstation memory sizes are increasing, however, a new technology trend is pushing toward small memories: mobile computers that are smaller than their desk-top counterparts and are typically configured with significantly less memory. Application designers are sometimes forced to squeeze their applications to fit into available memory, and may not succeed. Therefore, in a general-purpose mobile computer, as with many computers, paging is needed to enable a wider range of applications to run—as long as it can be performed efficiently.

The difficulty in paging on mobile computers arises from the same technology trend. While workstations are normally connected to relatively fast local-area networks and moderately fast disks, mobile computers may communicate over slower wireless networks and run either diskless or with small, slower local disks. At the same time, however, the processors on mobile computers are steadily improving in speed, and the disparity between processor speed and I/O speed is at

[*]This paper appeared in the Winter 1993 USENIX Conference, January 1993, pp. 519–529. Copyright to this work is retained by the author. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. This paper also appeared as Technical Report MITL-TR-18-92 (Revised).

[1]DECstation is a trademark of Digital Equipment Corporation.

least as great for mobile computers as for workstations. This disparity suggests a new technique for managing memory, which exploits *compression* to reduce I/O.

Compression is already widely used to reduce demand for secondary storage and networks. I suggest that it is now feasible to use compression to reduce the demand for memory as well. The basic idea is to take some memory that would normally be used directly by an application, and use it instead to hold a larger number of pages in compressed format. I call the area used for compressed data a *compression cache.* If the pages touched by a process could not normally fit in memory, but could fit into memory when some were stored in the compression cache, then the processor would never have to write a page to backing store (onto a local disk or over a network to another computer). Even if pages must be written to backing store, compressing them beforehand reduces the amount of data transferred.

The potential benefits of the compression cache depend on the relationship between the speed of compression and the I/O bandwidth of the system, as well as the compression ratio (anywhere from barely over 1:1 to about 4:1 in the experiments reported below). If the cost of compressing and copying a page were negligible, and pages compressed well, the compression cache could be used to give a computer the appearance of having additional physical memory. In practice, compressing and copying have costs associated with them, and the benefit of reducing traffic to the backing store is offset by the overhead of the compression cache. Overhead comes not only from the compression itself but from the additional page faults an application will experience when some memory is used for compressed pages (as well as the data structures used to support compressed pages). Furthermore, as mentioned above, not all applications compress well: for poorly suited applications, the effort to compress memory will be wasted and degrade rather than improve performance. Thus, depending on the application and the hardware environment, the benefits of reduced I/O may outweigh the costs of compression and additional faults, or vice-versa. Configuring the compression cache to improve performance in the first case while staying out of the way in the second case is an interesting, and difficult, problem.

The remainder of this paper is organized as follows. Section 2 discusses related work involving paging or compression. Section 3 elaborates on the tradeoffs involved with compressed paging. Section 4 describes the design of the compression cache, based on these tradeoffs. Section 5 evaluates the performance of the compression cache for some sample applications. Finally, Section 6 concludes the paper.

2 Related Work

This section discusses other projects and products with goals similar to the compression cache. They fall into two general categories: file systems and virtual memory.

2.1 File Systems

A number of systems have replaced individual users' *ad hoc* techniques for manual compression with a mechanism for automatically compressing some or all files. Cate and Gross used compressed files as a level in a hierarchy, with recently-accessed files being in uncompressed format and less-recently-used ones compressed [5]. Since frequently-used files were never compressed, and compression was performed in the background, the overall impact on interactive performance (delays due to decompression) was minimal: less than 50 seconds per user per day. At the same time, disk space requirements were roughly halved.

Burrows *et al.* integrated compression with Sprite LFS [12], also primarily to reduce disk space requirements [4]. They argued that LFS is a better vehicle for compressing files than traditional file systems, since files are not overwritten in place and a change to one block within a file would not cause changes to compressed data later in the file. Multiple file blocks may be compressed as a unit, providing better compression than if each block were compressed separately using a dynamic compression algorithm such as LZRW1 [16]. Burrows *et al.* found that on-line compression halved disk space requirements, as in Cate and Gross's system, without the delays that could be incurred by decompressing a large file as a single unit. The system had an acceptable performance degradation when compression was performed in software, and was well-suited to hardware compression.

In addition, there is a family of products for personal computers that do both on-line and off-line compression for the purpose of reducing disk space usage. A discussion of these products is available elsewhere [4], so it is omitted here.

2.2 Virtual Memory

The focus of the above systems has been disk space rather than performance. Other projects have considered ways not only to reduce disk space demands, but also to improve performance, particularly in the area of virtual memory.

Taunton described a mechanism for compressing binary executables on Acorn personal computers, reducing disk space requirements and improving file system bandwidth [14]. Because compression of the executables was performed off-line, an especially effective (but slower) compression algorithm was available. Bandwidth improved because the cost of decompression was offset by the reduction in data transferred from the disk. As a result, the performance of program invocation improved.

Atkinson *et al.* at Xerox PARC, considered the use of compression in order to reduce the cost of paging over wireless links [2]. Such paging might be needed in an environment with mobile computing devices that are too small to have local disks, such as the "tabs" advocated by Weiser [15]. The PARC researchers concentrated on read-only data, such as executables, because of the space and time overhead of performing on-line compression. Executables would be stored and transmitted in compressed format, and cached on a mobile computer in compressed format to increase the number of such executables that could be cached. Again, because compression would be performed off-line, an asymmetric compression algorithm could be used that would give very good compression ratios (with a correspondingly high overhead) while decompressing quickly. These researchers did consider on-line compression as well, resulting in a suggestion (reported by Appel and Li [1]) that pages be compressed and retained in memory. This idea, which they did not pursue extensively, is the primary theme of this paper.

3 Design Considerations

Intuitively, the idea of trading processing (compression) for I/O is appealing: by and large, processors are improving in performance more quickly than I/O devices, especially disks.[2] If one can compress some pages so that they occupy little enough memory to permit all of a process's address space to reside in memory, it might be possible to avoid I/O to the backing store completely; the process would execute correspondingly faster. Note that this technique is fundamentally different from writing a dirty page into a file system that does compression, or a disk that does compression at the driver level, because compressed pages never have to go to backing store at all. Instead, compressed pages form an intermediate level in the storage hierarchy, between uncompressed pages and the backing store.

[2]Paging over a network rather than to a local disk is another issue. In some environments, it is more efficient to page over a 10-Mbps Ethernet to memory on a file server than to page to a local disk [9]. Some local-area networks, such as ATM networks (*e.g.*, Autonet [13]), provide bandwidth that is at least an order of magnitude greater than an Ethernet. However, for mobile computers on wireless networks, one can expect the disparity between processing and I/O to remain for some time.

Keeping compressed pages in memory does not obviate the need for a backing store, however. It is possible for the collective address space of all running processes not to fit in memory even after compression. And even if they fit, it might be desirable to move some "old" pages to backing store in order to have more memory available for actively-used pages. In either case, pages could be transferred to backing store in compressed format, reducing the demand for bandwidth. This technique would be similar to paging into a file system or disk that does its own compression. The differences are:

Reduced I/O. With the compression cache, some pages might be faulted upon before being written to backing store. Those pages would no longer need to be written.

Variable memory allocation. By making compressed pages an explicit part of the memory hierarchy, the system can dynamically vary the amount of memory used for uncompressed pages, compressed pages, and file blocks. This is necessary to avoid impacting applications that do not need to compress pages, as discussed below in Section 4.2.

Complexity and space overhead. These are better when compression is performed at the level of the backing store, rather than the VM system. Assuming a one-to-one mapping between VM pages and file blocks, transferring pages that are already compressed requires the VM system to cluster multiple compressed pages into a smaller number of file blocks. Also, the VM system must manage free space on the backing store at a granularity finer than individual file blocks. Considering that the file system may use compression regardless of its use for virtual memory, the extra overhead to manage the backing store may be wasted effort and wasted memory. This issue is discussed further below.

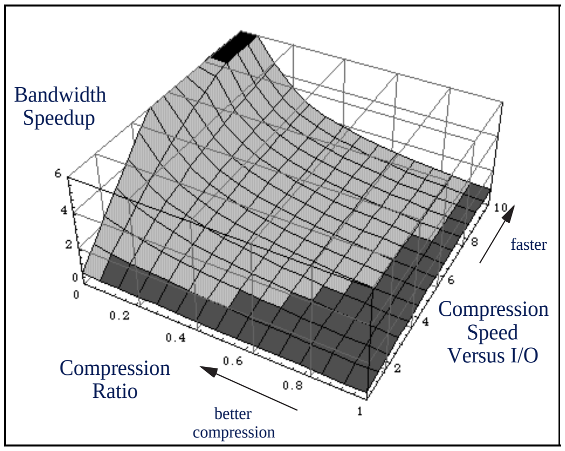
Regardless of whether compression is performed explicitly by the VM system or implicitly when pages are transferred to backing store, the effectiveness of compressing VM pages depends on several factors:

Compression speed. Compressing a page, and later decompressing it, must be significantly faster than transferring it to or from backing store. Otherwise, one might as well do traditional paging without compression.

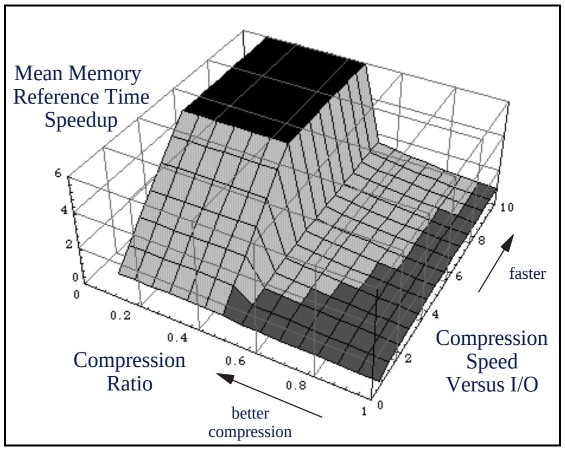
Compression ratio. On average, pages must compress to significantly less than their original size. Obviously, compressing a 4-Kbyte page to 3500 bytes is far less useful than compressing it to a few hundred bytes.

Page access patterns. If pages are compressed and retained in memory, then fewer pages are available for uncompressed pages. An application will likely take additional page faults, accessing pages that would be resident and uncompressed in a standard system but are instead stored in compressed format. Given this effect, it is important that the compression cache not degrade performance: if the collective working set of active processes fits into physical memory without the need to compress pages, the compression cache should stay out of the way. This implies that its size should vary dynamically over time as the demand for memory changes.

Memory overhead. Keeping pages in both uncompressed and compressed format has memory overhead associated with it (keeping track of the state of each page, as well as where pages are stored on disk). Taking away this memory from applications also results in additional page faults.



(a) Transferring compressed pages to backing store.



(b) Keeping compressed pages in memory.

Figure 1: Performance of compressing pages, modeled analytically. Speedups are shown as a function of the compression ratio (fraction of bytes left after compression) and the speed of compression relative to I/O. Decompression is assumed to be twice as fast as compression, as is roughly the case for algorithms such as LZRW1 [16]. There are three regions of speedup: the dark black areas at the top left show speedups that go off the top of the scale (6-fold improvement); the light areas show speedups of 1–6 relative to no compression, and the darker areas to the right show data points at which a slowdown would result.

Compression implementations. The compression cache should allow for both software- and hardware-based compression. Ideally, it should allow different compression algorithms to be used for different types of data, in order to get the best compression rates and/or throughput.

As one might expect, there is an inverse relationship between compression speed and compression ratio: the faster a page is compressed, the less compression is required for compression to improve performance. Figure 1(a) graphs the speed of paging to and from backing store in compressed format, as a function of compression bandwidth (relative to the bandwidth of the backing store) and compression ratio. Figure 1(b) shows the speedup of mean memory reference time as a function of these two variables, when pages are retained in memory, for an application that sequentially accesses twice as many pages as fit in memory, reading and writing one word per page. In this case, if pages are compressed to no larger than half their original size, on average, the speedup due to compression is linear in the speed of compression. Of course, if pages do not compress well, then compression must be much faster than I/O or overall performance will be worse than without compression. In some systems it is also possible for an application to issue an "advisory" to the operating system to indicate that least-recently-used (LRU) page replacement will behave poorly; in this example, half the pages could effectively be pinned in memory with faults occurring only on the other half. With fast compression, however, even reducing I/O by a factor of two will be inferior to keeping all pages compressed in memory.

The sharp leap in speedup when all pages fit in memory, as in Figure 1(b), demonstrates the potential difference between the compression cache and a system that compresses pages en route to the backing store. In practice, this improvement is not fully realized, because access patterns are not so pathological. The performance of sample applications is given after the description of a specific implementation of a compression cache for the Sprite operating system [11].

4 Design

This section describes the design and implementation of a compression cache in Sprite. Sprite is largely compatible with 4.3 BSD UNIX, but its virtual memory system has an interesting difference from most versions of UNIX: physical memory is traded dynamically between VM for application processes and the file system's buffer cache [9]. Since the compression cache must vary in size dynamically as well, Sprite provides a good framework for prototyping the compression cache. The idea of the compression cache should extend naturally to UNIX,[3] Mach, or other systems; in fact, Mach's external pager interface [7] should be an excellent foundation for future work in this area.

The target environment for this research consists of mobile computers with limited memory and network bandwidth, and with small local disks or no disks at all. Because of the limited availability of Sprite, the compression cache has been prototyped in a workstation environment, running on DECstation 5000/200 workstations, paging to a local RZ57 disk. The Sprite kernel is configurable at boot-time to allow the system to use a variable amount of physical memory, so a 32-Mbyte machine can behave as though it has as little as 12 Mbytes. About 6 Mbytes are used by the kernel for code, page tables, and some forms of tracing that cannot currently be disabled.

4.1 Overview

The compression cache forms a new level in the memory management hierarchy. A general description of the technique is as follows.

- LRU pages are compressed to make room for new pages. The compressed pages are retained in memory for a period of time, in the expectation that they will be accessed again soon.
- If not all pages fit in memory, even with some compressed, the LRU compressed pages are written to backing store.
- To service a page fault for a page that is not already uncompressed and resident in memory, the VM system checks to see whether the page is compressed in memory or on the backing store. If it is on backing store, it is first brought into memory and stored in the compression cache, then it is decompressed and made accessible to the faulting process. The compressed copy in memory can be freed at any time, since there is already a copy on backing store.

Specific issues arise in a number of areas. First, the VM system must be able to vary the amount of physical memory allocated to the compression cache, taking into account the demand for uncompressed pages and for the file system buffer cache. Second, the interface between the compression cache and the backing store is complicated by the notion of variable-sized pages. Finally, the overhead of managing the compression cache should not adversely affect performance. The following subsections discuss these issues.

4.2 Variable Memory Allocation

Initially, the compression cache was implemented as a fixed-size region of physical memory. This was done partly for simplicity and partly because the need to vary its size was not yet apparent. In this version, the compression cache consisted of a number of pages, each divided into N fragments (In my experiments, N was defined to be 8, meaning blocks of 512 bytes with a pagesize of

[3]UNIX is a registered trademark of UNIX System Laboratories, Inc.

4 Kbytes). When a page was compressed, the system allocated enough fragments to hold the compressed data. The fragments did not need to be allocated contiguously; instead, the compression was performed into a contiguous buffer and the compressed data was then scattered into multiple fragments. To satisfy a page fault, the fragments for a page were copied into a contiguous buffer and then decompressed.

The fixed-size implementation was simple, since unused fragments could be linked together on a list, and fragmentation could be kept to a minimum. But this implementation was suitable only for applications that paged heavily even without the compression cache, and which fit into the compression cache without excessive traffic to the backing store. For example, on a machine with 8 Mbytes of memory available to user processes, setting aside 4 Mbytes for compressed pages would cause a 6-Mbyte process to page, ruining its performance. On the other hand, even after compression a 12-Mbyte process probably would not fit into the 4 Mbytes available. In the first case, either no compression cache or a cache of less than 2 Mbytes would be better, but in the second case using 6–8 Mbytes for compressed pages might eliminate all traffic to the backing store.

The compression cache was therefore redesigned to vary its memory usage over time. At first I considered an extension of the previous design, with fixed-size page fragments, but there is a problem with this approach. To reclaim a whole physical page from the compression cache, to use for an uncompressed VM page or a file block, each fragment within the page must be copied elsewhere or written to backing store. Since a page with N fragments could contain a small piece of each of N different pages, either the physical page would be transferred directly to backing store (resulting in multiple I/Os to read all the fragments for a particular page upon a page fault) or several different pages would have to be transferred to backing store in order to free one physical page. In addition, the overhead of doing "scatter/gather" between the contiguous compression buffer and the page fragments is unnecessary.

Instead, memory for the compression cache is now treated as a variable-sized circular buffer. Physical pages are mapped into the kernel's virtual address space, one after another, eventually wrapping around to the start of the range of addresses for the compression cache. There is a notion of the *oldest* physical page—the one added to the cache the longest time ago—and new pages, which have been added most recently and may not contain data yet. Physical pages are added to one end of the queue and normally removed from the other end. (They may be removed from the middle if no clean pages are availabled at the oldest end.) When VM pages are compressed, they are compressed directly into the first unused region within the compression cache, following the last page that had been added to the cache. Before each page there is a small header that describes the page, the size it compressed to, whether it contains dirty data, a link to the next page in the cache, and other information.

Figure 2 shows a simplified view of the compression cache. Pages are in one of four states:

clean	A page that has had all modified compressed pages within it written to backing store. A page can also be clean if it contains only compressed pages that have been brought in from backing store to satisfy page faults.
dirty	A page with modified data that are not on backing store.
free	A slot in the compression cache that does not have a physical page associated with it.
new	A page not yet containing data. *New* pages can only exist at the tail of the queue.

Pages may also be reclaimed dynamically from the compression cache. The oldest page in the cache with unmodified data is unmapped and returned to the kernel's pool of free physical pages.

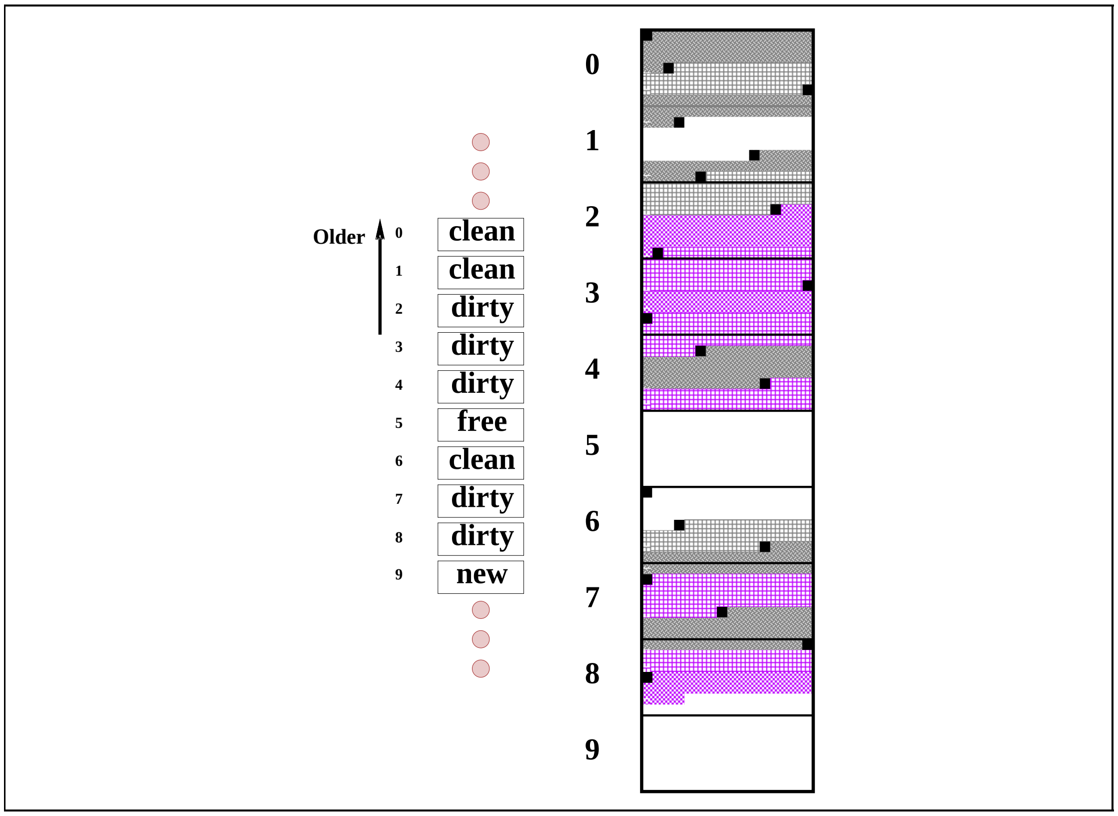


Figure 2: *State of the compression cache.* Physical pages may be in any of several states. An separate array of page descriptors stores the mapping of slots in the compression cache to physical pages and keeps track of the state of each page. Two stipple patterns represent distinct VM pages within the compression cache. Each user page has a small descriptor just before it indicating its state. Lighter pages are clean, while darker ones contain modified data. White areas contain no current data.

A kernel thread writes out the oldest dirty data in the compression cache in an attempt to keep a pool of physical pages clean and ready for reclamation. The rate at which pages are cleaned is a function of the number of completely free pages in the system, the number of clean pages that are already reclaimable, and the size of the compression cache.

The method for choosing when to grow or shrink the compression cache is similar to the algorithm in Sprite for trading memory between the file system and VM system. Sprite compares the age of the least-recently-used file block to the age of the LRU VM page, and reclaims the older of the two, modulo an adjustment to favor retaining VM pages longer. This "penalty" to the file system helps improve interactive performance, by preventing a large file from flushing a process's address space completely out of memory [10].

With the compression cache adding a third collection of pages, and a third consumer of memory, the tradeoffs are more complicated. In the current implementation, allocation of each of the three types of memory (file system cache blocks, uncompressed VM pages, and compressed pages) requires a comparison of the ages of the oldest pages for all three types. The system biases the ages to favor compressed pages over uncompressed pages and both of these over file cache blocks.

The more the system favors compressed pages, the larger the compression cache will tend to grow in periods of heavy paging; with a very low bias (or a bias in favor of uncompressed pages), the compression cache degenerates into a buffer for compressing and decompressing pages between memory and the backing store.

Interestingly, although a single penalty between VM and the file system works well across a wide range of applications, the optimal penalty for the compression cache is application-dependent. An application that exhibits a great deal of locality should have as many pages uncompressed at once as possible; thus the compression cache should serve just to buffer I/O to and from the backing store, but would not be expected to eliminate the I/O completely. Also, a large application that exhibits so little locality that its faults are rarely satisfied within the compression cache will not benefit from a large cache. Only an application with characteristics that cause it to "hit" in the compression cache will benefit from a large cache. Examples of such an application appear in the next section.

4.3 Interface to the Backing Store

In an unmodified Sprite system, the size of a VM page is an integral multiple of a 4-Kbyte file system block. Since on the DECstations (where the compression cache was prototyped) a VM page is 4 Kbytes, this discussion assumes a one-to-one mapping between file blocks and VM pages. When a page is written to backing store, it is written to a "swap file" corresponding to the segment containing the page, at an offset corresponding to the location of the page within the segment. This fixed mapping of pages to file blocks makes it trivial to locate a page on the backing store.

There are a number of ways to transfer variable-sized compressed pages to and from backing store, none of which is especially appealing. Ideally, the system would keep each compressed page in the same location in its swap file as without the compression cache, but transfer just the amount of data occupied by the compressed page. Unfortunately, with the exception of the last block in a file, the file system enforces transfers in multiples of a whole file system block. If part of a block is written then the file system reads the old contents and overwrites the part just written before writing the whole block back to disk. In other words, if a page were compressed from 4 Kbytes to 2 Kbytes, a 2-Kbyte write would result in a 4-Kbyte read and a 4-Kbyte write rather than only the expected 2 Kbyte write! Furthermore, a request to read 2 Kbytes within a 4-Kbyte block would result in the file system reading all 4 Kbytes and then copying just the part requested into the requesting process's buffer.

Without changing the internal structure of the Sprite file system, or writing every page into its own file (with significant overhead of its own), there is no way to avoid reading a minimum of 4 Kbytes to satisfy a page fault. This has the unfortunate effect of reducing the usefulness of the compression cache for applications that read a large number of pages in an unpredictable order: each page fault will require both a full 4-Kbyte read and a decompression. There are, however, possible solutions to the extra overhead for partial *writes* described above:

- A partial solution would be to issue an operation to write an entire block, thus writing 4 Kbytes but not first issuing a disk read. However, this would not benefit from having already performed compression. In an environment in which few pages are written to backing store this would be unimportant, but not all applications fit in memory even when compressed.

- Another possibility would be to modify the file system to overwrite part of a file system block on disk without reading the remainder of the block. In this case disk bandwidth would improve, but it would still suffer from having independent small I/Os rather than a small number of large I/Os. (Note that it might be possible to page into Sprite LFS [12], which provides much higher bandwidth by coalescing many small writes into a single larger transfer, but LFS suffers from the same restriction of 4-Kbyte transfers.)

- The solution I implemented attempts to transfer exactly the amount of data a page occupies when compressed, by merging several compressed pages into a smaller number of file blocks. This reduces fragmentation, with a corresponding reduction in bandwidth needs and disk space usage.

Merging compressed pages together has its own problems, however. First, this scheme loses the one-to-one mapping between offsets in a swap file and pages within a segment. Instead, it is necessary to store the location of each page explicitly. Second, when a page is written out to backing store, faulted back into memory, modified, and written out again sometime later, it may not be written to the same location. (If it were, the same problem of writing partial file blocks would occur.) Thus it becomes necessary to perform garbage-collection on the backing store to keep track of which blocks contain the most recent copy of page and which blocks contain obsolete data. If compressed pages can be written to arbitrary locations within a block, keeping track of the location and size of each page becomes a bookkeeping nightmare. Third, if pages are allowed to span two file blocks, it becomes necessary to read in both blocks to satisfy a page fault. Thus a 4-Kbyte `read` becomes an 8-Kbyte one. If page accesses exhibit sufficient locality that retrieving 8-Kbytes of compressed pages satisfies additional page faults without more I/O, spanning pages is not disadvantageous in the long run, but without this locality the system will pay a performance penalty.

The version of the compression cache I have implemented in Sprite pads each compressed page to a uniform fragment size (currently 1 Kbyte), and writes a set of fragments, spanning several file blocks, in a single operation. Currently 32 Kbytes of compressed pages are written at once. The system is parameterized to determine whether pages are allowed to span file block boundaries: if they cannot, then fragmentation increases and the effective bandwidth for writes to the backing store correspondingly decreases.

4.4 Overhead

The compression cache adds some overhead in terms of memory usage. The kernel sets aside a static buffer that is used for the LZRW1 algorithm's hash table [16]. This hash table can be relatively large (*e.g.*, on the order of 1 Mbyte), which improves compression at the cost of memory, or be relatively small. In the system measured for this paper, the hash table is 16 Kbytes. In addition, the difference in code sizes between the unmodified system and the system with the compression cache is an additional 22 Kbytes.

When a segment is created or enlarged, its page table is essentially extended by 8 bytes per 4-Kbyte page, which is used by the compression cache. While this is only 0.2% overhead for pages that are resident in memory, this information is resident even when pages are not: for non-resident pages, an unmodified system stores just 4 bytes per page, rather than 12 for the compression cache. As an example, if the collective virtual memory of all running processes is 60 Mbytes, with 4-Kbyte pages, the per-page overhead for the compression cache would total 120 Kbytes.

There is also overhead for the space managed by the compression cache itself. The kernel uses 8 bytes per page in the range of addresses the compression cache might occupy (as shown in Figure 2). This overhead is determined at boot time based on the maximum possible size of the cache. The kernel also allocates a 24-byte header within each physical page frame that is mapped into the cache (0.6% overhead), and a 36-byte header for each virtual page that has been compressed and placed in the cache. These overheads occur only when the compression cache has data in it, and are offset by the savings in memory usage due to compression.

5 Performance

As Figure 1 showed, the improvement due to compression depends on the speed of compression, the amount of compression obtained, and the number of tranfers to or from backing store that can be completely eliminated. I consider both the maximum possible performance improvement and the performance of some applications.

5.1 Maximum Possible Improvement

It is possible to estimate the maximum possible improvement for a particular configuration and compression algorithm by running an program that is contrived to thrash the VM system. *Thrasher* cycles linearly through a working set, reading (and optionally writing) one word of memory on each page each time through the working set. The system uses an LRU algorithm for page replacement, so if *thrasher*'s working set does not fit in memory, then it takes a page fault on each page access. If *thrasher* is modifying pages as it accesses them, the system must write a page each time to make room for the page being faulted on. The unmodified Sprite system, which uses regular files as the backing store, would perform two disk seeks for each fault, one to write a page out and another to retrieve the page faulted upon. If *thrasher* is only reading each page, then no seek is necessary if the pages are close to each other in the swap file (equivalently, close to each other in the address space).

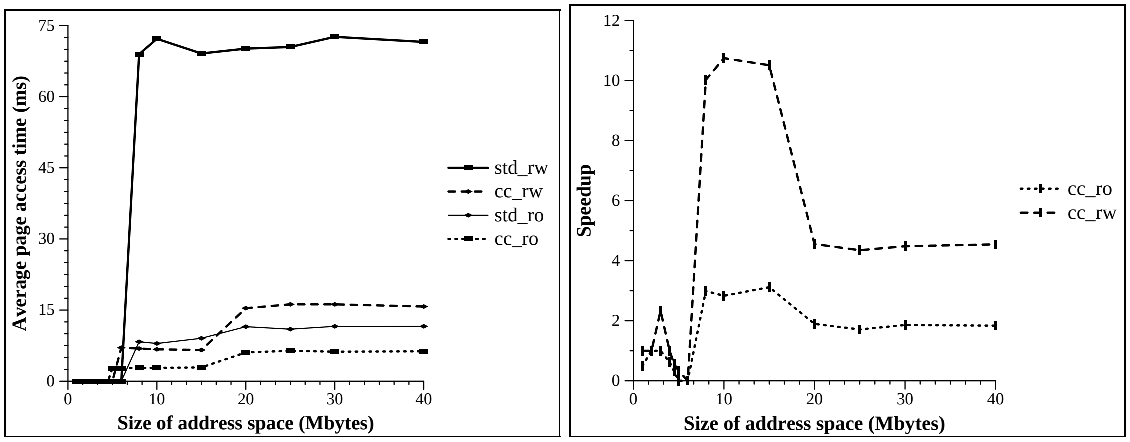
With the compression cache, *thrasher* would still fault on each page, but each fault could be satisfied by a decompression (and a compression, if pages are being modified) rather than one or two disk I/Os. Since taking some memory for compressed pages does not increase the fault rate—*thrasher* was faulting on every new page anyway—the ratio between compression speed and I/O speed determines the speedup. If the working set does not fit in memory when compressed, then each fault may require a read from the backing store, as well as possibly a write to make room for it. By clustering compressed pages together, however, transfers are effectively smaller, and multiple pages can be obtained with a single read from the backing store. This can reduce the average number of seeks per page fault considerably.

Note that Sprite LFS could alleviate the problem of seeks between pageouts by grouping multiple pages into a single segment. However, it is not clear that paging into LFS would be desirable under heavy paging load. LFS requires significant memory for buffers, and for LFS to clean segments containing swap files, it must copy more "live" blocks than for other types of data [12].

Figure 3 shows access time as a function of working set size, on a machine configured to use no more than 12 Mbytes (of which about 6 Mbytes are available to user processes). There are four lines:

std_rw	An unmodified Sprite system, sequentially reading and writing each page.
cc_rw	A Sprite system with a compression cache, sequentially reading and writing each page.
std_ro	An unmodified Sprite system, sequentially accessing each page without modification;
cc_ro	A Sprite system with a compression cache, sequentially accessing each page without modification;

Figure 3(a) gives the average time to access each page, and Figure 3(b) gives the speedup of the compression cache relative to the original system.



(a) Average page access cost. (b) Speedup relative to original system.

Figure 3: *Compression Cache Performance Under Thrashing.* With an unmodified system, a large number of pages fit in memory without measurable page-fault overhead, but once the system starts thrashing it pays for one or more disk operations per page access. With the compression cache, pages compress roughly 4:1. Compression reduces the average access time considerably, especially when compressed pages fit in memory without the need for disk I/O (up to a total address space of about 15MB). Larger address spaces, from 20 Mbytes upward, resulted in disk I/O, but with fewer transfers and fewer seeks than the unmodified system. Measurements were taken on a DECstation 5000/200 with approximately 6 Mbytes available for user processes, paging to a local RZ57 disk, with a page size of 4 Kbytes. Compression was performed using Williams's LZRW1 algorithm. The labels are explained in the text.

5.2 Application Speedup

The speedup of *thrasher* gives an upper bound on the performance improvement real applications might experience through the compression cache. This is because *thrasher* almost always takes the same number of page faults even when some memory is set aside for compressed pages, and the memory *thrasher* accesses compresses extremely well. Real applications may not compress as well, and they often exhibit a degree of locality that significantly increases their page fault rate as they are allocated less memory. This section reports the performance of some sample applications. A summary of these results is in Table 1.

One example of an application that exhibits a substantial improvement from the compression cache is a program that computes the sequence of modifications to change one file into another. *Compare* could be useful for transferring "diffs" rather than entire files, when the changes between two versions of a file are minimal. Lopresti implemented file differencing using a dynamic programming algorithm (refer to Lipton and Lopresti [8] for a description). The application uses a two-dimensional array, of which only a wide stripe along the diagonal is accessed. It works its way through the array in one direction, and then reverses direction and goes linearly back to the beginning. Elements along the diagonal are based on a recurrence relation that causes frequent repetitions in values, which in turn suggests that the data in the array are extremely compressible. With LZRW1 the pages compress about 3:1, and with other compression algorithms, the pages

Application	Time (std)	Time (CC)	Speedup	Compression Ratio (%)	Uncompressible pages (%)
compare	16:14	6:04	2.68	31	0.1
isca	43:15	27:00	1.60	32	1.7
sort_partial	13:32	10:24	1.30	30	49
gold_create	14:03	15:38	0.90	59	42
gold_cold	45:30	56:36	0.80	60	10
sort_random	26:17	28:51	0.91	37	**98**
gold_warm	35:56	49:00	0.73	52	0.9

Table 1: *Application speedups.* Measurements were taken on a DECstation 5000/200, paging to a local RZ57 disk with a page size of 4 Kbytes. All benchmarks were run with approximately 14 Mbytes available for user processes. Compression was performed using Williams's LZRW1 algorithm. Some applications had a high fraction of pages compress less than the threshold (4:3), though the applications with the greatest speedup compressed well across all pages. Times are in *minutes:seconds*.

should compress even better.

Another example of an application that benefits from the compression cache is Dubnicki's cache simulator, which is both CPU-intensive and memory-intensive [6]. In a sample run, *isca* experienced a 50% improvement in execution time, and pages that were compressed during its execution averaged a 3:1 compression ratio as well.

Despite these two examples of applications with good compression ratios and consequently good performance using the compression cache, applications do not necessarily compress especially well, and their performance suffers accordingly. I considered an application that performs `quicksort` on a file containing approximately 12 Mbytes of text (numerous copies of each word in `/usr/dict/words`). If the text were completely unsorted to begin with (*sort_random*), so there was minimal repetition of strings within an individual 4-Kbyte page, the *sort* program ran significantly more slowly on the compression cache than the unmodified system—primarily because about 98% of the pages compressed less than 4:3, the threshold for keeping them in compressed format. Thus the time to compress these pages was wasted effort. For the sake of comparison, *sort*'s heap compressed much better if the input file contained frequent repetitions of words—for example, if the input file were only a minor permutation of the sorted copy of the file, with substrings (or complete words) often repeated within a page of memory (*sort_partial*). In this case the compression ratio was about 3:1 and the application ran 23% faster than on the unmodified system (rather than 10% slower).

Finally, one might expect that a main-memory database would benefit from the compression cache if it fits in memory when compressed but not otherwise. Some accesses would be to data that tends to remain uncompressed ("warm" data), while others would be to less frequently used ("cold") data, which would stay mostly compressed. Each access to compressed data would incur the overhead of decompression (and subsequent compression if the page is modified), but not a disk I/O. However, the hit rate on uncompressed data would be lower than the hit rate in a system without the compression cache, because some memory would be used for compressed pages instead of regular virtual memory. The poorer the compression ratio, the greater the penalty.

Indeed, one such database, the "index engine" for the Gold Mailer [3], compresses slightly worse than 2:1; it runs more slowly under the compression cache than on an unmodified system. This is partly due to the poor compression and partly due to the high fraction of nonsequential page accesses it encounters, each of which requires a full 4-Kbyte read from backing store. Ideally, one would use the compression cache in a system that permitted less than a 4-Kbyte read to satisfy a page fault, in which case Gold (and other applications) should benefit more generally from compression. Table 1 lists three runs of *gold*:

gold_create	This benchmark creates a new index from scratch. It has a high degree of write accesses, so the degradation it suffers by reading 4-Kbyte blocks is partly offset by writing compressed pages together to backing store. However, 42% of pages compress less than 4:3, and the average of the rest is only 59%. The program runs 11% more slowly with the compression cache than without.
gold_cold	This benchmark performs a sequence of queries against an existing *gold* index engine, with the index engine having just started. Thus the index engine writes many pages as well as reading them. It runs 25% more slowly.
gold_warm	Lastly, this benchmark performs the same set of queries once *gold_cold* has executed. The index data are already established in the address space of the index engine, and are faulted upon in a read-only fashion. A small number of pages are modified as the program operates, however. This benchmark runs 36% more slowly.

Obviously, for those applications that run 20–40% more slowly with the compression cache, varying the amount of memory is insufficient to prevent degradation. It should be possible to disable compression completely when poor compression is obtained.

6 Conclusions and Future Work

In conclusion, compression can reduce the amount of I/O to and from a backing store, possibly eliminating it completely. Even when I/O operations are still necessary, compressed pages require less bandwidth. Depending on the cost of compression, the cost of I/O, and the compressibility of memory pages, this technique can improve performance by factors of 3–4 or more in the best case.

However, "real" applications generally do not obtain this degree of improvement for a number of reasons:

- locality, which causes an application to take faults on compressed pages that would have been accessible in an unmodified system;
- poor compressibility, which results in less of a reduction in I/O for the same amount of effort; and
- restricted I/O, which causes larger transfers to be performed than are necessary.

One example of an application that *does* obtain significant speedup is a file comparison program that compresses well and whose sequential passes through a large two-dimensional array make it less susceptible to an increase in the fault rate. Other applications vary from moderate improvements in performance to slight (or even substantial) degradations. As compression gets faster relative to I/O, the range of applications that can benefit from compressed paging should improve. This can happen in any of several ways: hardware compression, which would improve the disparity between compression speeds and I/O rates; faster processors, which would do the same thing for software compression; and slower backing stores, such as wireless networks. A better interface to the backing store would help as well.

Note that the same techniques presented in this paper for virtual memory can potentially be applied to other areas as well. For instance, on systems with enough physical memory to make Sprite LFS practical, one might consider combining compressed Sprite LFS [4] with the compression cache techniques presented here: the system could keep part or all of the file buffer cache in compressed format in order to improve the cache hit rate. One might also redesign specific applications, such as databases, to keep some of their data structures in compressed format, using application-specific techniques for compressing data and managing the choice of data to compress. Experiences with the compression cache make it clear that the success of any scheme that uses compression to improve performance will depend a great deal on the relative speeds of compression and I/O, the compressibility of data, and data access patterns.

Acknowledgements

Brian Marsh and Rafael Alonso contributed to the design and initial implementation of the compression cache, and helped to formalize its expected performance. Krish Ponamgi helped with the implementation of the backing store. Cezary Dubnicki, Steven Johnson, Dan Lopresti, and Karin Petersen provided programs to help evaluate the compression cache.

Rafael Alonso, Lisa Bahler, Daniel Barbará, Brian Bershad, Eben Haber, Hank Korth, Kai Li, Dick Lipton, Dan Lopresti, Brian Marsh, Krish Ponamgi, Sreedhar Sivarkumaran, and Marvin Theimer provided helpful feedback on earlier drafts of this paper, which helped improve its content and presentation. Lastly, I would like to thank Andrew Appel, Mike Burrows, Ramón Cáceres, Mike Jones, Karin Petersen, and Jonathan Sandberg for their comments and suggestions.

References

[1] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, CA, April 1991.

[2] Russ Atkinson, Dan Greene, Bryan Lyles, and Marvin Theimer. Applying compression techniques to virtual memory paging. Xerox PARC Internal Memorandum, 1990.

[3] Daniel Barbará, Chris Clifton, Fred Douglis, Hector Garcia-Molina, Stephen Johnson, Ben Kao, Sharad Mehrotra, Jens Tellefsen, and Rosemary Walsh. The Gold mailer. In *9th International Conference on Data Engineering*, Vienna, April 1993. To appear.

[4] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. In *The Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–9. ACM, October 1992.

[5] Vincent Cate and Thomas Gross. Combining the concepts of compression and caching for a two-level filesystem. In *The Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 200–211. ACM, April 1991.

[6] C. Dubnicki and T. LeBlanc. Adjustable block size coherent caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 170–180, Gold Coast, Australia, May 1992. ACM.

[7] David B. Golub and Richard P. Draves. Moving the default memory manager out of the Mach kernel. In *Proceedings of the 2nd Mach Symposium*, pages 177–188, November 1991.

[8] R. J. Lipton and D. P. Lopresti. Comparing long strings on a short systolic array. In W. Moore, A. McCabe, and R. Urquhart, editors, *Systolic Arrays*, pages 181–190. Adam Hilger, Boston, 1987.

[9] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[10] M. N. Nelson. *Physical Memory Management in a Network Operating System.* PhD thesis, University of California, Berkeley, CA 94720, November 1988. Available as Technical Report UCB/CSD 88/471.

[11] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[12] Mendel Rosenblum and John Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992. Also appears in Proceedings of the 13th Symposium on Operating Systems Principles, October 1991.

[13] M. Schroeder, A. Birrell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Satterthwaite, and C. Thacker. Autonet: A high-speed self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318–1335, October 1991.

[14] Mark Taunton. Compressed executables: an exercise in thinking small. In *Proceedings of the USENIX 1991 Summer Conference*, 1991.

[15] Mark Weiser. The computer for the 21st century. *Scientific American*, pages 94–104, September 1991.

[16] Ross N. Williams. An extremely fast ZIV-Lempel data compression algorithm. In *Data Compression Conference*, pages 362–371, April 1991.