Adaptive Compressed Caching: Design and Implementation[*]

Rodrigo S. de Castro[†], Alair Pereira do Lago[†], and Dilma Da Silva[§]
[†]Department of Computer Science
Universidade de São Paulo, Brazil
[§]IBM T.J. Watson Research Center, USA
rcastro@ime.usp.br, alair@ime.usp.br, dilma@watson.ibm.com
http://linuxcompressed.sourceforge.net

Abstract

In this paper, we reevaluate the use of adaptive compressed caching to improve system performance through the reduction of accesses to the backing stores. We propose a new adaptability policy that adjusts the compressed cache size on-the-fly, and evaluate a compressed caching system with this policy through an implementation in a widely used operating system, Linux. We also redesign compressed caching in order to provide performance improvements for all the tested workloads and we believe it addresses the problems faced in previous works and implementations. Among these fundamental modifications, our compressed cache is the first one to also compress file cache pages and to adaptively disable compression of clean pages when necessary.

We tested a system with our adaptive compressed cache under many applications and benchmarks, each one with different memory pressures. The results showed performance improvements (up to 171.4%) in all of them if under memory pressure, and minimal overhead (up to 0.39%) when there is very light memory pressure. We believe this work shows that this adaptive compressed cache design should be actually considered as an effective mechanism for improvement in system performance.

1 Introduction

Compressed caching is a method used to improve the mean access time to memory pages. It inserts a new level into the virtual memory hierarchy where a portion of main memory is allocated for the *compressed cache* and is used to store pages compressed by data compression algorithms. Storing a number of pages in compressed format increases effective memory size and, for most workloads, this enlargement reduces the number of accesses to backing store devices, typically slow hard disks. This method takes advantage of the ever increasing gap between the CPU processing power and disk latency time, which is currently about six orders of magnitude slower to access than main memory. This gap is responsible for, among other things, an underutilization of the CPU when the system needs exceed the available memory. An example of this effect is the Linux kernel compilation. Even when many processes of the compiler are run to compile the kernel source tree, the CPU usage drops substantially if the available memory is not enough for its working set. And this is also true for typical current systems with many hundreds of megabytes of memory experiencing heavy loads, such as web servers, file servers and database systems. In these scenarios, compressed caching can make a better usage of CPU power reducing accesses to the backing stores and smoothing performance drops when the available memory is not enough. The concern with these scenarios when the available memory is not enough is still adressed in current operating systems by improving their virtual memory systems, in particular their page replacement policies.

Although the reduction of accesses due to the compression tends to improve system performance, the reduction of *non-compressed memory* (main memory not allocated for the compressed cache) tends to worsen it. This inherent tradeoff leads us to question of how much memory which should be used by compressed cache, which depends on the workload to achieve the best performance. A compressed cache that adapts its size during the system

[*]This research was supported by FAPESP (Fundação de Amparo à Ciência do Estado de São Paulo), through grant 01/01432-4, and CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), through grant 465901.

execution in order to reach a good compromise is called *adaptive* and one with fixed size is called *static*.

The use of compressed caching to reduce disk paging was first proposed by Wilson [20, 21], and Appel and Li [2]. Douglis implemented an adaptive compressed cache in the Sprite operating system, achieving speedups for some experiments and slowdowns for others [4].

Given the inconclusive results of Douglis, the problem has been revisited by many authors. Russinovich and Cogswell implemented a static compressed cache in Windows 95 [18], which resulted in negative conclusions for the Ziff-Davis Winstone benchmark. On the other hand, Kjelso et al [7, 8, 9] empirically evaluated main memory compression, concluding that it can improve system performance for applications with intensive memory requirements. Kaplan [5, 22] demonstrated through simulations that a compressed cache can provide high reduction in paging costs. His experiments confirm Douglis' first statements about the limitations of a static compressed cache system. Moreover, he proposes an adaptive scheme that detects during system execution how much memory the compressed cache should use. This scheme warranted benefits for all the six programs he simulated using memory traces. An implementation of a static compressed cache in the Linux operating system was performed by Cervera et al [3]. In spite of the inherent limitations of a static compressed cache, they showed performance improvements for most of the tested workloads.

In this paper, we reevaluate adaptive compressed caching through real applications and benchmarks on an implementation in the Linux operating system. This implementation has been made public and tested by many people in different equipments for many months. We demonstrate that an adaptive compressed cache can provide significant improvements for most applications with a design that minimizes its costs and its impact on the virtual memory system as well as uses the allocated memory efficiently. Our implementation uses a new adaptability policy that attempts to identify at run time the amount of memory the compressed cache should use to provide the best compromise. This policy adds minimal memory and CPU overhead to the system.

We also show that the use of compressed caching alters the behaviour of various parts of the operating system and that its costs are beyond memory page (de)compressions. In particular, bad compression ratios for memory pages are shown to be manageable and do not affect compressed caching performance so severely as stated in previous studies. Furthermore, the adaptive compressed caching is revised in a period when the main reason that motivated this idea, the gap between CPU processing power and disk access times, has never been so wide.

In the following section, we describe the design of the compressed cache we implemented and Section 3 presents our analysis about static compressed caches as well as description of the adaptability policy we propose. Section 4 contains the experimental results of our implementation and Section 5 compares this implementation with previous works. In Section 6 we point out possible extensions to this work and present our conclusions.

2 Design

In this section, we first show an overview of the main concepts behind compressed caching and our implementation. Second, we discuss the overhead side effects. At last we give more details and some important design decisions in our implementation in the Linux 2.4.18 kernel.

2.1 Overview

In a compressed caching architecture, the main memory is divided into non-compressed memory and the compressed cache. When the virtual memory system decides to make room for new allocations, it evicts some pages, and they are compressed and stored in the compressed cache. Although it is possible for the page to be stored uncompressed when it is incompressible by the compression algorithm, throughout this paper, any page stored in the compressed cache is said to be a *compressed page*, regardless of the form it is stored in, and any page in the non-compressed memory is known as *non-compressed page*. Any attribute that a non-compressed page had at the moment it was evicted is inherited by the compressed page (for example, dirtiness).

Previous studies and implementations have designed compressed caches which only stored pages backed by swap. Unlike them, in our implementation all the pages backed by a backing store (e.g., file cache pages) are eligible to be compressed and stored in the compressed cache. The reason behind this decision will be discussed in Section 2.3.1.

In our implementation design, as soon as there is no space available in the compressed cache to insert a new page, either the compressed cache allocates more memory for its usage or some compressed pages are freed. The decision on which action should be taken will be discussed in Section 3. Later in this section the former action (allocating more memory for its usage) is described. When the latter action (freeing compressed pages) is taken, the oldest[1] compressed page is released. However, before being released, compressed dirty pages must first be written to

[1]The order the compressed pages are freed is in accordance to the order they were stored into the compressed cache.

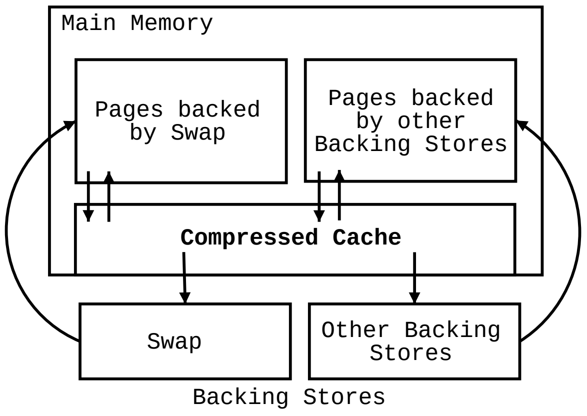

Figure 1: Memory Hierarchy with Compressed Caching

the backing store. Depending on the data they hold, compressed dirty pages may have their data decompressed before undergoing the write operation. In our approach, only pages backed by swap are written in compressed form, but we do not have swap space gains since each compressed page is null-padded to complete the size of a block, as to avoid swap space fragmentation, which would possibly be a source of overhead. Storing these pages onto the swap device in compressed form delays the decompression to the "swapin" operation and avoids decompression of pages never to be reclaimed by the system.[2] Compressed pages not backed by the swap must be decompressed before being written since a file system assumes that the data will be stored in its natural form.

Compressed pages requested back by any kernel operation in order to be immediately used are said to be *reclaimed*. This includes a page reclaimed by a page fault and a page holding data of a block cached in memory. Reclaimed compressed pages are removed from the compressed cache, decompressed, and their data are placed in newly allocated memory pages. If a compressed page was eventually back stored and is not present in the compressed cache (nor in the non-compressed memory), it is read from the backing store, and decompressed if the backing store is the swap. In this case, it is not added to the compressed cache when read from the backing store. See Figure 1 for the complete hierarchy.

An adaptive compressed cache must allocate memory in such a way that permits its memory usage to change during system execution. For this reason, we provide a simple infrastructure for management of the memory space allocated for compressed caching through paging.

[2]Pages that are read in advance ("read-ahead") are only decompressed if any process faults in them, i.e., they are mapped back by a process page table.

In the compressed cache, the smallest amount of memory that can be allocated or deallocated is known as a *cell*. A cell is formed by a constant number of *contiguous memory pages* and is used to store one or more compressed pages. It is important to notice that two consecutively allocated cells do not have necessarily contiguous addresses.

The *final free space of a cell* is the contiguous region at the end of the cell that does not store any compressed page. Whenever a page is compressed into the compressed cache, we search for the cell with the smallest final free space where the compressed page can fit and store it at the beginning of the final free space region. When a page is freed from the compressed cache, either because the compressed cache is full or because the page was requested by any kernel operation, it is simply removed from the cell in which it was stored. To avoid unnecessary overhead, no movement of compressed pages inside the cell is performed when pages are added to or removed from the compressed cache, which makes fragmentation in the cell a possible issue. The *free space* of a cell consists of the sum of space in all regions in the cell not used to store any compressed page. See Figure 2.

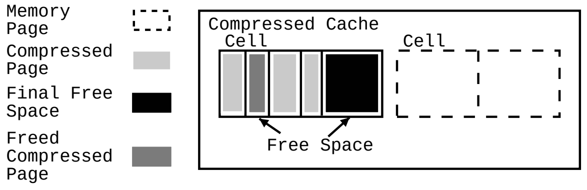

Figure 2: A cell in compressed cache

Depending on the compressed cache utilization, we may be unable to find a cell whose final free space is large enough to store a new compressed page, but there may be a cell whose free space is sufficient. In this case, a cell with the smallest free space where the compressed page can be stored is selected. Then, this cell is *compacted*, i.e., all compressed pages are moved to the beginning of the cell, making all free space available as final free space. Before enlarging the compressed cache or releasing any compressed page, we always try compaction.

2.2 Overhead Considerations

The time spent by compression algorithms is our primary overhead concern. Besides the time to compress and decompress a single page, which will be discussed later (see Section 4.4), the number of times the system (de)compresses pages must be taken into account. This amount depends on how much memory the processes use, their access patterns, and the memory usage of the compressed cache at a given time. There may be cases where, even if we succeed in reducing the accesses to the backing store, the total time spent (de)compressing pages can be substantial, notably when many (de)compressions occur. We may also detect some overhead when there are processes ready to run in the CPU most of the time, even if many I/O operations are performed concurrently. In this particular case, if all the (de)compressions use more than the CPU idle time, this usage may penalize running processes, slowing down their execution.

Since compressed caching allocates an amount of memory for storing compressed pages, it decreases the available memory that can be directly mapped by process's page tables. For this reason, the number of page faults tend to increase. Page allocations also tend to increase for two reasons: *(i)* more pages are needed to service the increasing number of faults; *(ii)* fewer blocks are likely to be cached, thus more allocations are needed to provide pages to blocks that are cached and uncached more often (see Section 2.3.1). As a consequence, the overhead introduced by compressed caching is also composed of costs from handling page faults and page evictions. These costs, notably releasing pages, can be substantial.

Another important effect of the compressed caching is the *metadata overhead* it introduces. Every cell allocated for the compressed cache needs metadata[3] about the compressed page(s) it stores. Furthermore, each cell has metadata about the compressed pages it holds. Depending on the number of cells, i.e., the compressed cache size, and on how many pages are stored in it, the memory space used by those data structures may be quite significant. A compressed cache system should also take into account the metadata costs its implementation requires. For this reason, simpler algorithms and strategies may be more effective, particularly if we consider that the system will be used under memory pressure.

In Linux, pages may be read from or written to a block device using auxiliary structures named buffers, which store block data in pages from the page cache (page cache is explained in Section 2.3.1). When buffers are used, they are marked as clean or dirty instead of the memory pages containing their data (I/O operations that do not use buffers mark the page holding their data itself as clean or dirty). In the page release process, pages holding data from dirty buffers must first be written to backing store before being eligible to be freed. Besides, given that the number of page allocations is higher with compressed caching (as described above), many more dirty buffers may have to write their data to allow the pages storing them to be freed. Therefore, compressed caching, in an attempt to reduce reads from backing store, may increase the number of write operations. As a matter of fact, we noticed this behaviour in some of the workloads tested. However, the impact of write operations on system performance is usually lower than of read operations. In particular, our experiments did not have their performance hurt by an increase in the number of write operations. Initially, we added support for storing pages holding dirty buffers data, but later we removed it for a number for reasons: *(i)* it achieved almost no performance gain; *(ii)* these pages could not be compressed due to the buffer handling code; and *(iii)* the support implied undesirable changes to the compressed cache structure.

[3]In comparison to a compressed cache of the same size and cells composed of one memory page, only half the data structures for those metadata is needed when cells with two contiguous memory pages are used.

2.3 Design Decisions

In this section, we give more details and some important design decisions in our implementation.

2.3.1 Page Cache

All previous studies proposed or implemented compressed caches that stored only pages backed by swap. When a compressed cache like these is used, all system caches end up being smaller since they have less available memory to compete for. In Linux specifically, system caches are namely the disk cache, which is known in Linux as *page cache*, and kernel internal data structure caches, known generically as *slab caches*. Examples of slab caches for kernel internal data structures are the buffer, inode, dentry and quota caches.

Compressed caching has a strong tendency to influence the page cache, as it is commonly larger than other caches. Pages holding data from blocks of all backing stores (like buffer data, regular file data, file system metadata and even pages with data from swap) are stored in the page cache. Like other system caches, page cache may be smaller on a system with a compressed cache that only stores pages backed by swap. As a consequence of this possible reduction, blocks (usually from regular files) will have fewer pages with their data cached in memory, what is likely to increase the overall I/O. That is a sign that compressed caching should not only be aware of its usefulness to the virtual memory system, but also how it might degrade system performance.

Instead of letting page cache and compressed cache compete for memory, our approach for this problem consists of also storing other pages from the page cache (besides the ones holding swap data) into the compressed cache. This actually increases memory available to all pages in page cache, not only to those backed by swap.

In Section 4, we show some tests where this design decision is shown to be fundamental.

2.3.2 Page Ordering

In the compressed cache, our primary concern regarding page ordering is to keep the compressed pages in the order in which the virtual memory system evicted them. As we verified in experiments on Linux, which uses an least recently used (LRU) aproximation replacement policy, not keeping the order in which the compressed pages are stored in the compressed cache rarely improves system performance and usually degrades it severely.

As most operating systems, when a block is read from the backing store, Linux also reads adjacent blocks in advance, because reading these subsequent blocks is usually much cheaper than reading the first one. Reading blocks in advance is known as *read-ahead* and the blocks read ahead are stored in pages in non-compressed memory.

Read-ahead operations alter the LRU ordering since the pages read in advance are taken as more recently used than the ones stored in the compressed cache, although they may even be not used. As a consequence, it is possible that this change forces the release of pages not in conformity to the page replacement algorithm. For this reason, whenever a page is read from the compressed cache, a read-ahead must not be performed.[4] It is not worthwhile to read pages from the compressed cache in advance since there is no performance penalty for fetching the pages in different moments. Furthermore, compressed pages read ahead from swap (which are stored in compressed form, as described in the Section 2.1) are only decompressed when explicitly reclaimed by the virtual memory system.

In contrast to the pages read only due to the read-ahead operation, a compressed page reclaimed for immediate use preserves LRU page ordering, since it will be more recently used than any page in the compressed cache.

We also consider essential to preserve the order in which the pages were compressed to be able to verify the efficacy of compressed caching. Otherwise the results would be influenced by this extra factor, possibly misleading our conclusions.

2.3.3 Cells with Contiguous Memory Pages

We say that the *compression ratio* of a compressed page is the ratio of the size of the compressed page over its original size (times 100%). We employ *rich compressibility*, *poor compressibility*, or *quasi incompressibility* if the average compression ratio is smaller than 50%, between 50% and 70%, or over 70%, respectively.

[4]The usual behaviour of Linux for pages read from somewhere else but non-compressed memory is to perform a read ahead, because it assumes they will be read from a slow backing store

If we do not have rich compressibility, compressed cache cells composed of only one memory page may store only one compressed page, in average. To minimize this problem of poor compression ratios, we propose the adoption of cells composed of contiguous memory pages. With larger cells, it is more likely to have memory space gains even if most pages do not compress very well. For example, if pages compress to 65% in average, we will still have space gains if we use cells composed of at least two contiguous memory pages. In fact, in this case, it is possible to fit three compressed pages in one cell.

However, we should notice that allocating contiguous memory pages has some tradeoffs. The greater the number of contiguous pages, the greater the probability of failure when allocating them, given the system memory fragmentation. Furthermore, the larger the cell, the greater the probability of fragmentation in it and the cost to compact its compressed pages. As a good side effect, given that part of our metadata is used to store data about the cells, the use of larger cells reduces these data structures. Experimentally, we have concluded that two is the number of contiguous pages to be allocated that achieves the best results in our implementation.

In Section 4 we will see some tests where this design decision is fundamental.

2.3.4 Disabling Clean Page Compression

For some workloads, no compressed cache scheme can improve system performance. This occurs when very few or no pages are read from the compressed cache among all the pages that were compressed. In this case, it is clear that a large amount of pages were compressed and freed, without any actual benefit to the system. As a matter of fact, compressing these pages added inherent costs like compression, decompression, management, and metadata.

This scenario can theoretically also happen with dirty pages, but in our experiments we could only observe it with clean pages, mainly due to the support for all pages backed by backing stores (other pages but the ones backed by swap are usually clean). We were unable to find a realistic application whose dirty pages had this problem. Moreover, for clean pages this problem is clearly more evident since no backing store operation is performed and the compression costs are highlighted.

In our implementation, we adopt a heuristic to attempt to detect when clean pages are being compressed without benefit to the system. This heuristic tries to detect the scenarios where a large amount of pages are compressed, not requested back by the system, and freed, without benefit to the system. This attempt is done by checking the relation between how many clean compressed pages are reclaimed by any kernel operation and how many of them are released from the compressed cache without being reclaimed. Once the compressed cache detects that many more compressed pages are released than reclaimed, it disables compression of new evicted clean pages. From this moment on, evicted clean pages are freed from non-compressed memory without being first stored in the compressed cache. As soon as we stop compressing the evicted clean pages, the compressed cache keeps track of which were the last clean pages freed without being stored in the compressed cache. Enough data that identifies these last freed pages are stored and all the pages read from disk are verified to match any of these data. When the system notices that many pages read from disk have recently been evicted from memory without being compressed, we re-enable the clean page compression.

If we release clean pages without compressing them into the compressed cache, the LRU page ordering is changed because some of the pages freed by the virtual memory system will be stored into the compressed cache and others will not. Nevertheless, since few of the clean pages were being reclaimed by the system, most of them would be freed anyway. Hence, it is expected that releasing them earlier does not have a major impact on system performance. The metadata and processing overhead introduced by this heuristic are insignificant.

The parameters used in these detections (when we disable clean page compression and when we re-enable their compression) were decided experimentally. In Section 4 we will see some tests where this design decision is very important to minimize substantially the overhead.

2.3.5 Variable Compressed Cache Size

In our experiments, we analyzed static compressed caches in many cases, reaching conclusions about them and the significance of an adaptive compressed cache. Since this is a key issue in our work, we will discuss, in the following section, the design decisions related to it.

3 Adaptive Cache Size

We observed in experiments that, given a particular application, different sizes for static compressed caches achieve different cost/benefit ratios. Even the best ratio among all the static sizes does not necessarily mean that a static compressed cache provided performance gains in comparison to a system without compressed cache.

In fact, static compressed caches smaller than the one with the best cost/benefit ratio provide fewer gains reducing accesses to the backing store than the optimal size. Moreover, they still introduce the inherent overhead (see Section 2.2), and for this reason the system would have a smaller benefit from the compressed cache use. On the other hand, static compressed caches larger than the one with the best cost/benefit ratio provide more gains reducing accesses to the backing store. But it also introduces more overhead to the overall system due to the reduction of non-compressed memory (see Section 2.2). This overhead hinders it from improving performance proportional to the gains it is able to provide reducing accesses to the backing store(s). See static compressed cache cases in Figure 3 for an illustration of this behaviour.[5] The adaptive case in this figure will be explained later in this section.

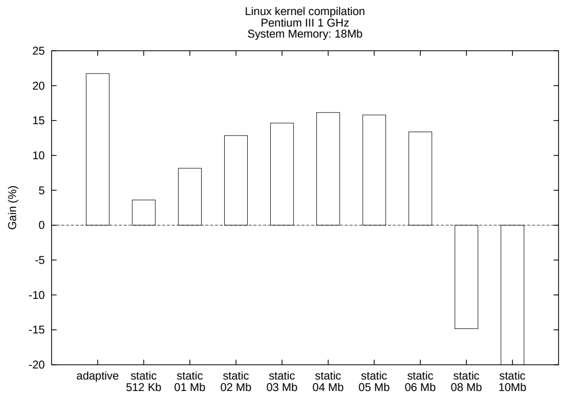

Figure 3: Comparison of several compressed caches with a kernel without compressed caching, showing the relative gains of total time for Linux kernel compilation (*j1*). These relative results were obtained for the adaptive compressed cache e for static compressed caches with sizes ranging from 512Kb to 8Mb.

In particular, the compressed cache cost is highlighted when the memory allocated for it is barely used. In fact, it introduces overhead to the system by reducing the amount of available memory for the non-compressed memory, but does not provide all the benefits that could be achieved with a compressed cache of this size.

In spite of the limitations mentioned above, static compressed caches may still improve application performance. However, even if a static compressed cache with a certain size is able to improve system performance for a specific application, given that programs have different memory needs during their execution, it does not mean that this static compressed cache can provide performance improvements for other applications. Furthermore, a static compressed cache with a particular size that improves the performance for an application may even

[5]Throughout this text, Kb means 1024 bytes, Mb means 1024Kb and Gb means 1024Mb.

degrade severely the performance for other ones. These conclusions are in accordance with the analysis already reported and/or predicted by previous work [4, 18, 5].

A reliable detection, at run time, of how much memory should be kept compressed in the system is the greatest challenge for compressed caching to become a general-purpose solution for the reduction of accesses to the backing store.

We designed and implemented an adaptability policy for the compressed cache to attempt to detect when it should change its memory usage in order to provide more benefits and/or decrease its costs. Our approach is based on a tight compressed cache, without allocation of superfluous memory. This improves memory usage efficiency, since unused memory does not improve performance, possibly even degrades it. The amount of memory allocated for compressed cache is only increased when it is full, compacted, and would have to release a compressed page in order to store a new page. The cells used by the compressed cache are released to the system as soon as they are no longer needed (e.g., when all compressed pages within a cell are freed). It is important to notice that, once we release a cell to the system, it is uncertain that we are going to be able to allocate it again when necessary, specially cells composed of more than one page. Page allocations and releases have the inherent cost of the functions that perform these tasks, but they are insignificant. This is particularly true because we make sure that the page allocations and releases of this scheme are performed without activating the virtual memory system to try to force releases, which is expensive. In other words, we only allocate if the memory pages we need are available at the allocation time.

In terms of how the compressed cache adapts its size to system behaviour, we begin with the assumption that compressed caching is useful to the system unless our online analysis shows otherwise. Hence, the cache starts with a minimum size at initialization time and, at first, it does not have limitations to its growth. That is, as soon as the virtual memory system starts to evict pages, the compressed cache increases its memory usage in order to store them.

Based on the approximated LRU ordering and on the actual compression ratio (how many pages are stored for the number of allocated memory pages), the pages in the compressed cache are split into two lists: *expense* and *profit* (see Figure 4). The expense list stores the compressed pages which would be in the memory if compressed caching were not used in the system. The profit list stores the compressed pages that are still in memory only due to the compressed caching.

The analysis that detects when the compressed cache

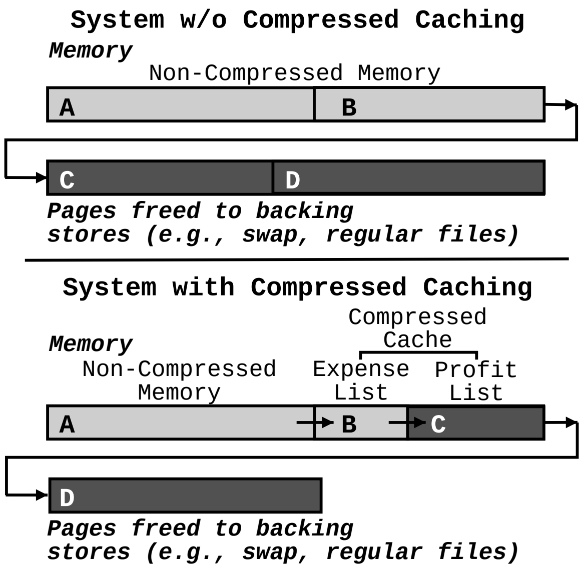
System w/o Compressed Caching
Memory
Non-Compressed Memory
A
B
C
D
Pages freed to backing stores (e.g., swap, regular files)
System with Compressed Caching
Compressed Cache
Memory
Non-Compressed Memory
Expense List
Profit List
A
B
C
D
Pages freed to backing stores (e.g., swap, regular files)

Figure 4: Lists in the compressed cache

has to change its size happens when pages are read from the compressed cache and is based on decompression of pages from the expense and profit lists (see Figure 5). In general, if compressed pages from the expense list are reclaimed to the non-compressed memory, our policy takes this as a sign that compressed caching is not being worthwhile, because these pages are being accessed with overhead. On the other hand, pages reclaimed from the profit list indicate that compressed caching is beneficial to the system, since these pages would have had to be read from disk if the system had no compressed caching.

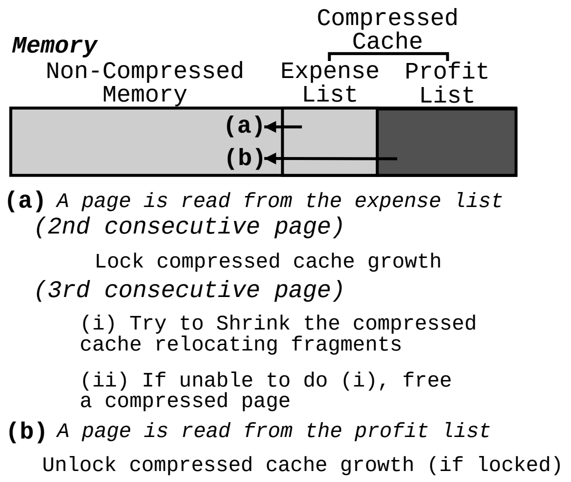
Compressed Cache
Memory
Non-Compressed Memory
Expense List
Profit List
(a)
(b)
(a) A page is read from the expense list
(2nd consecutive page)
Lock compressed cache growth
(3rd consecutive page)
(i) Try to Shrink the compressed cache relocating fragments
(ii) If unable to do (i), free a compressed page
(b) A page is read from the profit list
Unlock compressed cache growth (if locked)

Figure 5: Adaptability Heuristic

If two consecutive compressed pages are read from the expense list, the compressed cache is locked to growth, i.e., the compressed cache will no longer increase its memory usage. If this scenario persists and the third consecutive compressed page from the expense list is reclaimed, we take the following action: *(i)* if we are able to relocate enough of the compressed pages from a cell to other cells, we do the relocation, decreasing the compressed cache memory usage by freeing the initial cell; otherwise, *(ii)* we free the oldest compressed page. Once one of these actions is taken, the growth barrier is removed and the analysis starts again. If a page is read from the profit list, any existing growth barrier is removed (see summary of the heuristic in Figure 5).

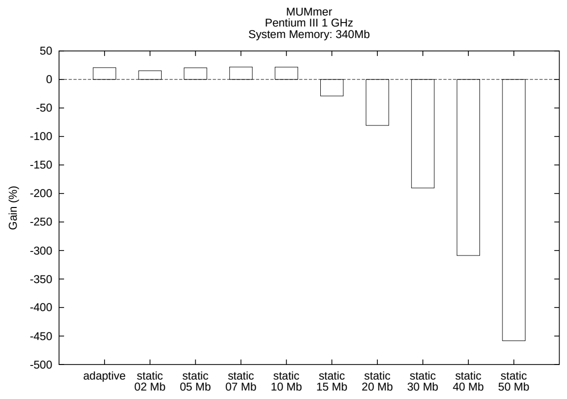

Figure 6: Comparison of several compressed caches with a kernel without compressed caching, showing the relative gains of total time for MUMmer execution. These relative results were obtained for the adaptive compressed cache e for static compressed caches with sizes ranging from 2 to 50Mb.

¿From our experiments, we verified that our adaptability policy achieves good results. When a static compressed cache provides performance gains in comparison to a kernel without compressed caching, our adaptability policy ends up selecting the size that achieves gains usually very close to the ones obtained by the best static size. Moreover, in some scenarios, applications may benefit from the adaptability policy (mainly avoiding superfluous allocated memory), achieving results better than any static compressed cache. See Figures 3 and 6 for results of a comparison relative a kernel without compressed caching between a static and an adaptive compressed cache.

4 Experimental Results

In this section, we describe the test suite, the methodology and the results of our implementation of a compressed cache in Linux 2.4.18.

The source code of the compressed cache implementation along with all the used tests programs and input data are available on our Web site [11]. The results presented in this paper were run with the 0.24pre5 version of our code.

4.1 Test Suite Description

Linux Kernel 2.4.18 Compilation [6] A very realistic benchmark with high CPU and memory usage, mainly when forking its build. The Linux kernel compilation was run as only one process (`j1`), as two (`j2`), and four (`j4`) concurrent processes. The memory available to the system varied from 18 to 48 Mb. The case with 48 Mb almost does not have memory pressure.

MUMmer 1.0 [15] A scientific application to align genomes that has a very intensive memory usage. Almost all of its memory usage is composed of pages backed by swap. MUMmer was executed varying the amount of memory size available to the system from 330 to 500 Mb. The 500 Mb case almost does not have memory pressure.

Open Source Database Benchmark (OSDB) 0.14 [16] A benchmark that performs several database related operations using the Postgres database manager and a 40 Mb database. We ran the experiments with 24 and 48 Mb of memory in the system. The 48 Mb case almost does not have memory pressure.

httperf 0.8 [14] A tool to measure web server performance. Our test using httperf performs a number of requests of a file to a local web server, measuring the number of resquest served per second. The available system memory varied from 24 Mb, which has a very intense memory pressure, to 64 Mb, where there is a very light memory pressure for this benchmark.

Matlab [13] A mathematical tool to perform numerical computations and graphics. In our tests, we used a Matlab program that computes the fractal dimensions of an image given as input. This computation may imply in an intensive memory and processing time usage. We used three different images and the virtual address space required by each of them was 80, 256 and 1000 Mb, respectively.

Sort - GNU textutils 2.0 [19] The sort program from GNU textutils aims at sorting a text file with minimal memory usage. It uses about 2 Mb of memory to sort files whose size are more than 100 Mb. The experiment results were collected with 24 Mb of memory in the system.

PostMark 1.4 [17] A benchmark to measure performance for small files in an operating system, by creating a large pool of files and measuring some transactions rates. This benchmark was run with 128 Mb of memory in the system.

4.2 Methodology

Our experiments were run on a system with a Pentium III 1 GHz processor, 768 Mb of system RAM and a 60 Gb, UltraDMA 100, 7200 rpm hard disk. The system had its memory configuration adapted for each application in order to force memory pressure and make compressed caching work. For most tests, an amount of memory under which the system almost does not have memory pressure is chosen.

The Linux distribution installed on this system was Debian Sarge. Each test was run after a fresh system reboot to avoid hot cache effects.

4.3 Compression Algorithms

We provided support for two compression algorithms in our implementation, namely:

WKdm [24] is a variant of WK compression algorithms family [5, 22] by Paul Wilson and Scott Kaplan designed for compressing in-memory data rather than file data. It uses a simple, direct-mapped dictionary of recently seen words.

LZO [12] is a fast Lempel-Ziv [25, 26] algorithm and implementation by Markus Oberhumer. In our implementation, we used the miniLZO, a lightweight version of the LZO library.

4.4 Performance Results

Global results. The results for the Linux kernel compilation for the various concurrency levels (`j1`, `j2`, and `j3`), the MUMmer, the Open Source Database Benchmark, httperf and Matlab are displayed in Table 1. In this table we present the execution time for a kernel without compressed caching (*w/o CC*) and various kernels with compressed caching. (The only two exceptions are httperf and Matlab, where only the data for the two most important kernels were collected.) Each compressed cache kernel has a different configuration (different adaptability policy, number of pages in a cell, compression algorithm or the type of pages it stores).

The *reference* column corresponds to the kernel with compressed caching that achieved the best results in our experiments. This means that it introduces improvements in most workloads, even if other configurations may improve even more in some cases. It uses LZO compression algorithm, cells composed of two memory pages and the adaptability policy described in Section 3. In comparison to the results from *w/o CC*, we observe significant gains in the scenarios with high memory pressure, reaching up to 171.4% of speedup. When it is under light memory pressure, a slight overhead (not more than 0.39%) occurs like noticed in the cases 30 Mb (kernel `j1`), 48 Mb (kernel `j4`), 500 Mb (MUMmer) and 48 Mb (OSDB).

We chose some amount of available memory to the system in order to verify compressed caching varying from almost no memory pressure to high memory pressure. For example, 500 Mb is more than enough to run MUMmer application, while 330 Mb is very close to degenerate the application behaviour intensively (with 320 Mb on a kernel without compressed cache, the MUMmer execution is almost 2 hours, i.e., about 4600% longer than with 330 Mb). As one can see for cases 27Mb of Linux kernel compilation (j1, j2, and j4), if the concurrency level increases, the memory pressure gets higher and compressed cache makes a huge difference (notably in `j4` case).

We also evaluated the overhead under no memory pressure at all. In order to do so, we run these tests on our system without any memory limitation, thus having 768Mb of RAM available. (For Matlab, we changed the input images to use less memory, as is seen in the 80 and 256Mb Matlab cases on the table.) For most cases (kernel `j2`, MUMmer, OSDB, Matlab (256Mb) and Matlab (80Mb)) no difference was noticed. For two cases (kernel `j2` and httperf), we noticed that compressed caching introduced an inherent overhead of 0.25-0.27%. For one case (kernel `j4`), we noticed that compressed caching achieved an improvement of 0.28%. We think it is fair to state that no overhead is expected by our implementation if the application is to be run with no memory pressure at all.

Compressing only pages backed by swap. Comparing columns *onlyswap* and *reference* from Table 1, we can see the importance of also compressing pages not backed by swap, as discussed in Section 2.3.1. For the MUMmer cases, one can see that *onlyswap* and *reference* achieve similar results. In fact, few pages are not backed by swap in these cases. Nevertheless, we see that the gain in OSDB 24 Mb case obtained by *reference* is nullified by *onlyswap*. The latter produces slowdown for most kernel compilation cases in contrast with the former, that achieves gains in most cases. If we run the kernel compilation tests for static compressed caches of several fixed sizes that store only pages backed by swap, the same slowdown is verified. It is also interesting to notice that kernel compilation and OSDB have higher usage of pages backed by other backing stores.

Cells composed of one, two and four memory pages. Comparing the columns *cell1*, *reference* and *cell4*, we see the influence of compressed caches with one, two and four pages in every cell, respectively. One can see that cells composed of four pages does not perform well in

test	memory	w/o CC	reference	sluggish	aggressive	onlyswap	wkdm	lzowkdm	cell1	cell4
	Mb	seconds	gain (%)	gain (%)	gain (%)	gain (%)	gain (%)	gain (%)	gain (%)	gain (%)
kernel (-j1)	18	467.8	21.73	18.07	21.46	-4.65	2.11	11.07	19.06	-0.55
	21	326.68	8.89	8.20	7.97	-1.01	3.20	5.71	7.33	1.64
	24	289.05	0.20	-0.35	0.66	-0.69	-1.44	-0.44	0.49	-1.56
	27	280.45	0.11	-0.64	0.17	-0.31	-0.44	-0.09	-0.19	-0.60
	30	278.33	-0.23	0.11	0.03	0.46	0.48	0.39	0.35	0.65
	48	274	0.18	0.09	0.28	0.47	0.31	0.33	0.26	0.34
	768	271.17	-0.27	-	-	-	-	-	-	-
kernel (-j2)	18	1002.62	33.13	11.68	31.76	1.60	0.36	0.30	28.86	-4.97
	21	608.98	33.84	21.99	30.40	-4.12	4.76	9.50	25.19	-4.41
	24	395.05	18.78	12.55	19.38	1.22	-0.18	11.26	15.59	0.43
	27	313.8	5.37	6.80	6.92	0.00	2.72	4.14	6.32	-0.53
	30	283.7	1.12	0.92	0.24	-0.00	-1.36	0.60	1.38	-0.23
	48	272.3	0.19	0.37	0.28	0.51	0.40	0.36	0.31	0.52
	768	269.76	-0.01	-	-	-	-	-	-	-
kernel (-j4)	18	1826.14	14.98	11.84	9.77	-0.52	6.73	5.01	13.31	-9.91
	21	1067.47	15.62	8.38	-1.50	-5.41	-5.67	-5.05	12.31	-11.72
	24	826.44	31.85	-2.16	20.60	-0.52	-1.73	-1.98	28.78	-7.65
	27	654.83	34.72	7.90	29.08	3.86	1.47	-6.64	33.09	2.01
	30	489.67	26.45	13.45	25.47	0.84	0.34	3.14	26.48	5.89
	48	274.95	-0.39	-0.84	-0.64	-0.16	-0.56	-0.70	-0.48	-0.66
	768	271.23	0.28	-	-	-	-	-	-	-
MUMmer	330	143.5	16.09	16.47	-120.36	-51.03	23.80	-5.93	37.08	-29.84
	340	115.21	20.74	21.95	14.47	22.18	23.06	25.23	38.64	13.49
	360	82.86	26.25	26.64	24.04	24.33	26.90	26.60	24.13	25.67
	380	81.21	16.71	15.66	13.98	12.26	15.66	16.75	38.75	19.10
	400	80.55	23.02	23.36	20.21	21.44	23.48	25.95	39.85	20.14
	420	58.51	15.11	14.19	14.10	16.34	20.18	20.15	14.80	10.48
	500	45.35	-0.22	-0.20	0.02	0.02	-0.26	-0.04	0.02	-0.07
	768	44.7	-0.09	-	-	-	-	-	-	-
OSDB	24	1242.4	30.70	31.59	31.91	-1.27	5.51	29.35	1.05	-7.69
	48	758.97	-0.07	-0.08	-0.63	0.75	0.30	0.08	-0.77	0.26
	768	735.5	0.00	-	-	-	-	-	-	-
Matlab (1Gb)	768	5880.36	6.12	-	-	-	-	-	-	-
Matlab (256Mb)	768	1977.83	-0.01	-	-	-	-	-	-	-
Matlab (80Mb)	768	579.30	-0.03	-	-	-	-	-	-	-

test	memory	w/o CC	reference	sluggish	aggressive	onlyswap	wkdm	lzowkdm	cell1	cell4
	Mb	reqs/sec	gain (%)	gain (%)	gain (%)	gain (%)	gain (%)	gain (%)	gain (%)	gain (%)
httperf	24	38.5	171.38	-	-	-	-	-	-	-
	32	117.7	153.40	-	-	-	-	-	-	-
	36	1529.1	14.10	-	-	-	-	-	-	-
	40	1849	1.40	-	-	-	-	-	-	-
	48	1646.1	14.95	-	-	-	-	-	-	-
	64	1819	3.20	-	-	-	-	-	-	-
	768	1894.1	-0.25	-	-	-	-	-	-	-

Table 1: Table with our results for a kernel without compressed caching (*w/o CC*) and several versions of kernel with compressed cache. *Reference* is the main compressed cache implementation and any other kernel differs from it in only one aspect. *Wkdm* uses WKdm compression algorithm instead of LZO, while *lzowkdm* uses LZO and WKdm. *Onlyswap* has a compressed cache that stores only pages backed by swap. *Sluggish*, and *aggressive* have different adaptability policies, but then *cell1* and *cell4* have cells composed of only one and 4 pages, respectively.

almost all cases, except for MUMmer. For kernel compilation tests, cells with one page performs as well as cells with two. In MUMmer tests, where we have rich compressibility, cells with one page perform even better in almost all cases, achieving gains of up to 39.85%. The only case where *reference* is clearly better than *cell1* is the case OSDB 24 Mb since we have poor compressibility (64.5%), as was discussed in Section 2.3.3. In this case, *cell1* speeds up its performance by only 1.05% and *reference* by 30.70%.

	LZO	**WKdm**
compression time	0.09 ms	0.05 ms
decompression time	0.04 ms	0.03 ms
compression ratio (kernel)	39.4%	60.6%
compression ratio (mummer)	35.5%	41.6%
compression ratio (osdb)	64.5%	86.5%

Table 2: Average time to (de)compress a page and average compression ratios for some tests per compression algorithm.

LZO vs WKdm. ¿From Table 2, we can notice that the WKdm compression algorithm compresses and decompresses a page faster than LZO, but it does not compress so tightly as LZO does. Comparing columns *wkdm* and *reference* in Table 1, one can verify that WKdm performs well in tests where the compression is not so worse than the one obtained by LZO. This is the case of MUMmer 330 Mb, where *wkdm* improves its performance by 23.80% and *reference* by 16.09%. Nonetheless, in all other tests, the compression ratio showed to be more important than the compression speed. In fact, a better ratio means that fewer pages will likely be freed and thus more reads from the backing stores will be saved. It should be said that MUMmer is a memory intensive application and all other applications depend much more on the page cache compression (compare columns *onlyswap* and *reference*). WKdm was designed to compress data segment pages and was not designed to compress other kind of pages like those backed by files. For this reason, we tried an implementation where LZO compression algorithm is applied to all pages except those backed by swap, which are compressed by WKdm (see column *lzowkdm*). In fact, this kernel performs better than *reference* for some cases of MUMmer, but its results are quite unsatisfactory for kernel compilations, in particular in the `j4` cases. This may be explained by the fact that the average compression ratio of swappable compressed pages is 36% with LZO against 50% with WKdm. Despite this unsatisfactory result, we still believe that one could obtain some improvement in this direction.

Other adaptability policies. Besides the adaptability policy we adopted (column *reference*), results from other adaptability policies are shown in Table 1: *sluggish* and *aggressive*. The first policy is less sensible to changes than *reference*, trying to take an action when there is stronger evidences. It provides smaller gains in the tests performed, except for some few cases. The second policy, on the other hand, is more aggressive, trying to shrink the compressed cache at each access to the expense list. This policy also provides smaller gains than *reference* in general.

Disabling clean page compression. In order to check the policy that disables clean pages compression, we performed experiments with the sort program from the GNU textutils 2.0 and also with PostMark benchmark. Both these applications have a substantial performance drop when compressed cache without the policy to disable clean pages compression is used. The sort program runs 40.9% slower, while postmark takes 49.4% longer to complete (see Table 3). In the latter, possibly because it is a benchmark, designed for other kind of comparisons, instead of a real application, only 0.06% of all pages in the compressed cache are read back by the system. PostMark's scenario is worse because the compression ratio of 99% (almost all pages are incompressible).

system	**sort**	**postmark**
CC - disabling clean pages compression	55.28s	329s
CC - not disabling clean pages compression	80.35s	529s
w/o CC	54.38s	330s

Table 3: Sort and postmark results disabling or not the clean pages compression.

With our policy that disables clean pages compression, the slowdown noticed running PostMark vanishes and sort takes only 1.6% more time to complete than a kernel without compressed caching. A minor slowdown is expected since the compressed cache takes some time to detect that the compressed cache is not worthwhile for clean pages.

Effect upon scheduling. A subtle consequence of compressed caching is its effect upon the process scheduling. In our opinion, this side effect is good since the CPU time sharing among processes is fairer if compressed caching is present as we will show in this section. In fact, in a kernel without compressed caching, whenever a fault on a page stored on a backing store occurs, a read operation is submitted to the disk. Since reading the page takes some time to complete, the read operation is submitted and the current process relinquishes the CPU. Then another process, if available, is scheduled to take control of the CPU. On the other hand, in a kernel with with compressed caching, if a page is stored in the compressed cache, this page will be decompressed to service any fault on it and unlike a fault on a page stored on a backing store, this service does not relinquish the CPU since this operation does not depend on slow devices, such as a hard disk.

Due to compressed caching, applications will likely have fewer faults on pages stored on the backing stores, therefore they will relinquish the CPU fewer times to service a page fault. If two or more applications are running on the system and compressed caching saves page faults of some of them which would generate reads from the backing store, these applications will run much faster than on a system without compressed caching. On the other hand, the applications that had less saved page faults may run even slower because they will not make use of the CPU time previously relinquished by the applications that have more saved faults.

As an example of this effect, we present an analysis of the behaviour of contest 0.51, a Linux kernel responsiveness benchmark [10]. Specifically, we discuss about the memory load test, which consists of running a Linux kernel compilation concurrently with a program (*mem_load*) that allocates 110% of the physical memory size and performs various accesses to this allocated memory. The main measurement of the benchmark is the time the Linux kernel takes to compile, but contest also displays how *mem_load* performed during the kernel compilation, through the number of iterations *mem_load* performed. After some experiments, we noticed that *mem_load* program faults on more pages on the backing stores than the kernel compilation process. Therefore without compressed caching, the compilation process takes advantage of the CPU idle time as a consequence of the scheduling performed by *mem_load* to wait for pages to be read. With compressed caching, *mem_load* does not have faults on pages stored on the backing stores since its data is highly compressible. Hence, the kernel compilation may take longer to run on a system with compressed caching, because *mem_load* uses a fairer CPU share, not relinquishing it to wait for pages to be read from the backing stores. As a matter of fact, *mem_load* runs much often with the compressed cache (see Table 4).

system	**kernel compilation completion time**	**number of iterations *mem_load* performs**
CC	94.90s	174
w/o CC	90.64 s	41

Table 4: Contest 0.51 benchmark results for *mem_load* load. Our testing system was set to run with 256 Mb of system memory (**CC** is compressed caching above).

5 Related Work

In Section 1, we presented a brief history and description concerning previous works, with or without implementations, on compressed caching. In this section, we compare as much as possible our implementation with those works. Moreover, from the understanding obtained with our own implementation we also contribute with some initial analysis on these works. We are mainly interested in the implementations by Douglis [4], Russinovich and Cogswell [18], Cervera et al [3] and also in Kaplan's adaptive scheme proposal [22, 5].

We tried to compare our implementation with the previous ones, but neither Douglis' nor Cervera et al's implementations made their test suite available. The Winstone [23] benchmark used by Russinovich and Cogswell is available only for Windows operating systems. Among the implementations, only the one by Cervera et al is available. However, we were unable to run some of our tests on a system with it, due to segmentation faults in kernel space (*oopses*, in Linux terminology). We were unable to find out if we hit new programming errors in their implementation or Debian Potato distribution tools were incompatible with the Linux kernel version they used.

The first implementation of compressed cache was performed by Douglis [4] in 1993, achieving inconclusive results: speedups for some applications (up to 62.3%) and slowdowns for others (up to 36.4%). He presented some reasons for these results, namely: *(i)* poor data compressibility; *(ii)* locality, that is responsible to make an application to fault on pages stored in the compressed cache which would be accessible without it; and *(iii)* restrictions in I/O operations that do not permit reads to benefit from compression by transferring smaller amounts of data. At last, Douglis observed that compressed caching will become more interesting if the gap between CPU processing power and disk access times keeps growing. Kaplan [22, 5], later in 1999, pointed out that the machine Douglis used to evaluate his implementation was many times slower than current computers, thus the gap between CPU and disks was much smaller than today. He also pointed out that the growth of the gap is a tendency in years to come. About the adaptive scheme devised and implemented by Douglis, Kaplan asserted that it may be maladaptive for many workloads and proposed a new scheme.

Experimentally, as Douglis, we noticed that poor compressibility was a problem for compressed caching. In our work we addressed this problem using cells with a greater number of contiguous memory pages (as seen in

Section 2.3.3). Regarding locality, in our implementation, any application that has many faults on pages that would be directly accessible without compressed cache (i.e., accesses on the expense list) and does not have enough benefits (i.e., accesses on the profit list) will force the compressed cache to shrink itself in order to adapt to a size which does not suffer from locality. We believe that the I/O restrictions Douglis mentioned do not hinder compressed caching from providing improvements, mostly because today we have disks with a high transfer rate, but still slow access times. In Douglis' implementation, the order the pages are stored in the compressed cache is not followed when the compressed cache is full and pages need to be freed. Therefore, the page replacement ordering is changed, what is likely to degrade performance. His compressed cache only stores pages backed by swap, but the adaptive scheme also takes into account pages not eligible to be compressed. Douglis also implemented support for storing more than one compressed page on a swap block, effectively increasing the swap space.

In 1999, Cervera et al [3] implemented a compressed cache in the Linux 2.0.36 as part of an implementation of compressed swap. They achieved performance gains for 7 out of the 8 tested benchmarks they reported. In comparison to our implementation, their compressed cache has a number of limitations. First of all, they implemented a static compressed cache, which is suitable for a small range of applications, as also noticed by Douglis and Kaplan. Their compressed cache is allocated with cells composed of single memory pages, thus they may have problems due to poor compressibility since a cell may not be able to store more than one compressed page. In their implementation, as Douglis', the order the pages are freed when the compressed cache is full does not follow the order they were stored in it. Their compressed cache stores only pages backed by swap which are marked as dirty, thus not compressing pages backed by other backing stores nor any sort of clean page. Moreover it is unclear how compressed swap and compressed cache separately influenced their results.

Russinovich and Cogswell could not obtain improvements with their implementation (neither with other commercial products like MagnaRAM 2 and RAM Doubler) for the Winstone on an Intel 80486 DX2/66 [18] and a slowdown of 10% was reported. In our opinion, there are three main reasons for these unsatisfactory results. The first one is that the used machine was slower and had a smaller gap between CPU power and disk access time than current machines, as was already pointed out by Kaplan. The second reason is that the obtained compression ratio was 62.5%. This is a poor compression ratio if compared to the applications and benchmarks we tested. If there is no specific support for poor compressibility, performance may drop substantially, as we have seen for the Open Source Database Benchmark when running with 24 Mb of system memory. The third reason is that they report a huge difference between the time spent on page (de)compression (0.05ms) and the time to service a page fault (2ms, without the actual seek time). In contrast, this difference almost does not exist in our experiments on Linux. They attribute this difference to a bad behaviour in the operating system used in their simulations (Windows 95). To conclude, a possible fourth reason is that the time spent on page (de)compression (0.05ms) itself seems to be too short, as already noticed by Kaplan. Despite our much faster machine, we obtained a close value only with WKdm. Since they do not report which compression algorithm they used, this could be a sign of a too fast but not effective compression algorithm.

Kaplan [22, 5] concluded that an implementation of an adaptive compressed cache that minimizes the overhead can provide significant reduction in paging costs. He proposed an adaptive scheme based on a cost/benefit analysis to detect the amount of system memory the compressed cache should use. This scheme was utilized in his simulations to demonstrate that the compressed cache can provide significant reduction in paging costs. In spite of proposing a compressed cache only for pages backed by swap, we think that Kaplan's adaptive scheme based on a cost/benefit analysis can be extended in order to detect the amount of memory that the compressed cache should use even if it stored all kinds of pages backed by backing stores. Nevertheless, it is very hard to collect the necessary data for the Kaplan's cost/benefit analysis efficiently on current systems. For instance, Linux does not keep the LRU list of all process data pages that are in memory. There is no such information and only an approximate part of this information is obtained when the system is under memory pressure. Furthermore, the extra memory required to store these data in the Linux kernel and consequently metadata overhead was not accounted in Kaplan's analysis. This may be discouraging since metadata overhead showed to have strong influence, particularly under memory pressure. In spite of these problems for Kaplan's cost/benefit analysis, his work has a strong contribution pointing out that previous inconclusive implementations could be corrected by a good adaptability policy for today's CPU/disk gap.

Our compressed caching aims at current standard hardware systems. Nevertheless, we should mention that some works designed and implemented compressed caching in hardware. Kjelso et al [7] designed and implemented in hardware a memory compressor. Using simulations, they demonstrated high performance gains. Abali and Franke [1] evaluated a system built for compressing *all* the main memory data, focusing on memory size increase. In their system, the whole main memory is compressed and a cache is used for storing decompressed data.

6 Conclusions and Future Work

We proposed a new and simple adaptability policy for a compressed caching system, implemented this system, and obtained significant performance improvements for all tested workloads under memory pressure (up to 171.4%) with negligible overhead under light memory pressure (up to 0.39%). Moreover, almost no overhead at all was detected if no memory pressure was applied. This compressed caching system is the first one to address workloads with poor compressibility and also to compress pages not backed by the swap device. These features showed to be fundamental for the improvement of performance in some workloads that could not otherwise obtain benefit from the use of compressed caching.

We also verified that the overhead introduced by metadata maintenance may have strong negative impact on system performance, particularly under memory pressure. Considering that a compressed cache system tries to obtain improvements right under this scenario, we observed that our simpler algorithms that require less metadata tend to obtain better results.

Improvements in the direction of a reduction of overhead under very light memory pressure were not tried. They are possible and should be tried in the future.

This implementation should also be extended to take advantage of multiprocessor architectures. Linux 2.4.18 does not give support to access process information about any evicted memory page. From this kind of information, one could try to perform an adaptive analysis that could also take a per process information into account. This could be helpful for workloads composed of different processes with different behavior.

Many users of our compressed cache system reported better responsiveness for desktop workloads. They reported smoother system interaction, what cannot objectively be evaluated yet, but is a strong indication of the benefits of adaptive compressed caching for common desktop workloads. We believe that good methods that test responsiveness under desktop workloads should be developed to scientificly confirm (or not) these very positive (but still subjective) reports. Considering that desktop workloads are the most important workload for most users, one can see how valuable are these reports and how important would be such a scientific confirmation.

We also believe that our compressed caching system will be extremely helpful on effectively extending memory on devices without swap like many PDA's that are able to run Linux. A specific comparison methodology for this application should be developed and applied.

One can state that memory is getting cheaper and compressed caching is then unnecessary. Based in our experiments, we claim that both, more memory and an adaptive compressed caching, are even better. Our experiments with applications for which all the installed memory is not enough showed that the obtained performance can save extra money by delaying another memory expansion.

To sum up, the main goal of the current implementation was achieved with the speedup verified in all workloads under memory pressure. For all these reasons we believe that an adaptive compressed caching must be adopted in current operating systems, as a mechanism for considerable improvement in system performance.

Acknowledgments

We would like to thank greatefully Fabio Kon and Livio B. Soares for the insightful comments and suggestions on this paper, Imre Simon for his support, Scott Kaplan, Marcelo Tosatti, Paolo Ciarrocchi, Marc-Christian Petersen and Con Kolivas for their comments and feedback from the implementation.

References

[1] B. Abali and H. Franke. Operating system support for fast hardware compression of main memory contents. In *Memory Wall Workshop of the 27th Annual International Symposium on Computer Architecture (ISCA-2000)*, Vancouver, BC, Canada, 2000.

[2] A. W. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, CA, USA, 1991.

[3] R. Cervera, T. Cortes, and Y. Becerra. Improving Application Performance through Swap Compression. In *Usenix '99 – Freenix Refereed Track*, 1999.

[4] F. Douglis. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *Winter 1993 USENIX Conference*, pages 519–529, 1993.

[5] S. F. Kaplan. *Compressed Caching and Modern Virtual Memory Simulation*. PhD thesis, University of Texas at Austin, 1999.

[6] The Linux Kernel Archives. `<http://www.kernel.org>`.

[7] M. Kjelso, M. Gooch, and S. Jones. Main memory hardware data compression. In I. C. S. Press, editor, *22nd Euromicro Conference*, pages 423 – 430, September 1996.

[8] M. Kjelso, M. Gooch, and S. Jones. Empirical study of memory data. In IEE, editor, *IEE Proceedings Comput. Digit. Tech.*, volume 145, pages 63 – 67, January 1998.

[9] M. Kjelso, M. Gooch, and S. Jones. Performance evaluation of computer architectures with main memory data compression. *Journal of Systems Architecture*, 45:571 – 590, 1999.

[10] C. Kolivas. The homepage of contest, The linux kernel responsiveness benchmark. URL: `<http://contest.kolivas.net>`.

[11] Compressed caching. `<http://linuxcompressed.-sourceforge.net/>`.

[12] Markus F.X.J. Oberhumer: LZO data compression library. `<http://www.oberhumer.com/opensource/lzo/>`.

[13] The MathWorks - MATLAB. `<http://www.mathworks.com/-products/matlab/>`.

[14] D. Mosberger and T. Jin. httperf A Tool for Measuring Web Server Performance. `<http://www.hpl.hp.com/personal/-David_Mosberger/httperf.html>`.

[15] The MUMmer Home Page. `<http://www.tigr.org/-software/mummer/>`.

[16] The Open Source Database Benchmark. `<http://osdb.-sourceforge.net/>`.

[17] PostMark: A New File System Benchmark. `<http://www.-netapp.com/tech_library/3022.html>`.

[18] M. Russinovich and B. Cogswell. RAM Compression Analysis. Technical report, O'Reilly, 1996.

[19] GNU Textutils 2.0 source code. `<ftp://ftp.gnu.org/gnu/-textutils/textutils-2.0.tar.gz>`.

[20] P. R. Wilson. Some Issues and Strategies in Heap Management and Memory Hierarchies. In *OOPSLA/ECOOP Workshop on Garbage Collection in Object-Oriented Systems*, 1990.

[21] P. R. Wilson. Operating System for Small Objects. In *Internation Workshop on Object Orientation in Operating Systems*, pages 80–86, Palo Alto, CA, USA, 1991. IEEE Press.

[22] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Summer 1999 USENIX Conference*, pages 101–116, Monterey, CA, USA, 1999.

[23] Business Winstone. `<http://www.etestinglabs.com/-benchmarks/bwinstone/bwinstone.asp>`.

[24] WKdm Compression Algorithm source code. `<http://-www.cs.utexas.edu/users/oops/compressed-caching/-WKdm.tgz>`.

[25] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

[26] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.