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 in order to improve system performance through reduction of accesses to the backing stores. We propose a new and simple 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 tested workloads and address the problems faced in previous works and implementations that led to non-conclusive results. Among these fundamental modifications, our compressed cache is the first one to also compress file cache pages, to adaptively disable compression of clean pages when necessary and to address applications with poor compressibility.

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 higher to access than main memory and tends to increase. 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. 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.

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 the question of how much memory should be used by compressed cache, whose appropriate size to achieve the best performance is dependent on the workload. A compressed cache that adapts its size during the system 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 [19, 20], and Appel and Li [1]. Douglis implemented an adaptive compressed cache in the Sprite operating system, achieving speedups for some experiments and slowdowns for others [3]. Given the incon-

clusive results of Douglis, the problem has been revisited by many authors. Russinovich and Cogswell modeled a static compressed cache for Windows 95 [17], which resulted in negative conclusions for the Ziff-Davis Winstone benchmark. On the other hand, Kjelso et al [6, 7, 8] empirically evaluated main memory compression, concluding that it can improve system performance for applications with intensive memory requirements. Kaplan [4] 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 [2]. In spite of the inherent limitations of a static compressed cache, they showed performance improvements for most of their test suite.

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 machines for many months. We demonstrate that an adaptive compressed cache can provide significant improvements for most applications with a design that minimizes costs and impact on the virtual memory system as well as uses the allocated memory efficiently. Our implementation uses a new and simple adaptability policy that attempts to identify at run time the amount of memory to be used by the compressed cache to provide the best compromise. This policy adds minimal memory and CPU overhead to the system.

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. Finally 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, which are compressed and stored in the compressed cache. Any attribute that a page from non-compressed memory had at the moment it was evicted is inherited by the compressed page (for example, dirtiness).

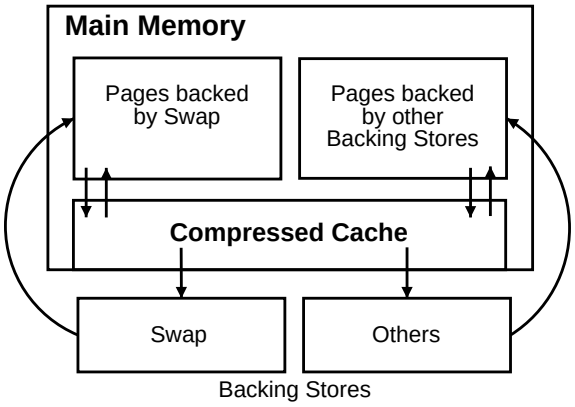

Figure 1: Memory hierarchy with compressed caching.

In our implementation, unlike previous ones, all the pages backed by a backing store (e.g., file cache pages) are eligible to be compressed and stored in the compressed cache. Also, as soon as there is no available space 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. When the latter action is taken, the oldest compressed page is released. However, before being released, compressed dirty pages must first be written to the backing store. Only pages backed by swap are written in compressed form, and each compressed page is null-padded to complete the size of a block. 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. Pages that are read in advance ("read-ahead") from swap are only decompressed if any process faults in them, i.e., they are mapped back by a process page table.

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 block data cached in memory. Reclaimed compressed pages are removed from the compressed cache, decompressed, and their data are placed in newly allocated memory pages. When a reclaimed page is not present in the compressed cache, it is read from the backing store, and decompressed if the backing store is the swap. The page is not added to the compressed cache when read from the backing store. See Figure 1 for the complete hierarchy.

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. The *free space* of a cell consists of the sum of space in all regions in the cell that are not used to store any compressed page. See Figure 2 for an overview of cell's structure.

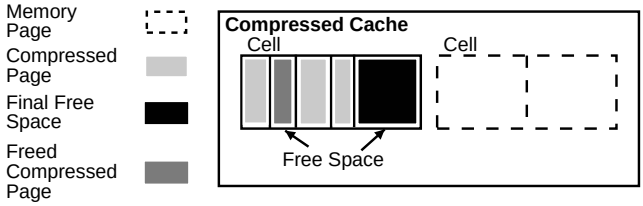

Figure 2: A cell with 2 memory pages where 4 compressed pages were stored and the second one was released later. The total free space includes the final free space and the space of the freed compressed page.

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 total 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 be adjacent in 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

Our primary overhead concern is the time spent to compress and decompress a single page and the total number of times the system (de)compresses pages. In particular, 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 and page allocations tend to increase. Another reason for the latter one is that fewer blocks are likely to be cached in memory, resulting in a higher number of data being cached and uncached.

Another important effect of the compressed caching is the *metadata overhead* it introduces. Compressed cache needs metadata for every allocated cell, and every cell needs metadata for 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. For instance, in our implementation, a compressed cache with 32Mb that stores in average 8 compressed pages in a 8Kb cell, we have about 2 Mb of data structures (for every 32768 stored compressed pages and every one of the 4096 cells, among other metadata costs). Moreover, the impacts on the Linux particular implementation must be considered an overhead consideration, like the effect upon the VM and buffers. For this reason, simpler algorithms and strategies may be more effective, particularly if we consider that the system will be used under memory pressure.

2.3. Design Decisions

In this section, we list important design decisions in our implementation, which will be shown to be fundamental in Section 4. The most important design decision, the adaptive cache, is detailed in Section 3.

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 the file cache, known as *page cache* and usually the largest one, and kernel internal data structure caches (*slab caches*). As a consequence of the possible reduction of page cache, blocks (usually from regular files) will have fewer pages with their data cached in memory, what is likely to increase the overall I/O. Instead of letting page cache and compressed cache compete for memory, our approach for this problem consists of also storing pages from the page cache into the compressed cache. This actually increases the amount of memory available to all pages in page cache, not only to those backed by swap.

Page ordering. Our concern regarding page ordering is to keep the compressed pages in the order in which the virtual memory system evicted them. Therefore, the compressed page chosen to be freed from the compressed cache when it is full (and is locked grow, accordingly to our adaptability heuristic) is the oldest one in the compressed cache. This page ordering was not respected in previous implementations. For example, in these works, any compressed page located in the same cell where the oldest page chosen to be freed might also be freed before other older pages. Besides that, Linux standard behavior for read-ahead operations induces programming mistakes which turn out to change page ordering and were carefully taken into account during development.

Cells with contiguous memory pages. When 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 adopted cells composed of two contiguous memory pages. With larger cells, it is more likely to have memory space gains even if most pages do not compress very well. We notice that allocating contiguous memory pages has some tradeoffs, like greater probability of allocation failure. Furthermore, the larger the cell, the greater the probability of fragmentation and the cost of compaction.

Disabling clean page compression. In our implementation, we adopt a heuristic to attempt to detect when clean pages are being compressed without benefit to the system, disabling or enabling clean page compression accordingly. 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.

3. Adaptive Cache Size

We observed in experiments that, given a particular application, different sizes for static compressed caches achieve different cost/benefit compromise. A static compressed cache with a particular size that improves the performance for an application may even degrade severely the performance for other ones. See static compressed cache cases in Figure 3 for an illustration of static compressed caches behavior.[1] The static compressed cache with best compromise in this example is the one with 4Mb. The adaptive case in this figure will be explained later in this section.

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. 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). 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 force releases, which is expensive.

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. As soon as new pages are compressed, pages will also move from the expense to the profit list.

1 Throughout this text, Kb means 1024 bytes, Mb means 1024Kb and Gb means 1024Mb.

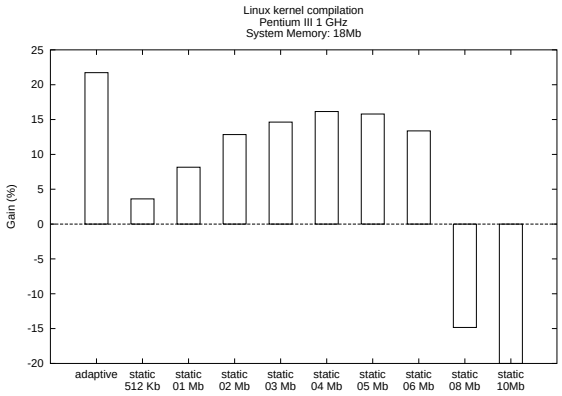

Figure 3: Several compressed caches were compared against a kernel without compressed caching. Relative gains of Linux kernel compilation total time (*j1*). These relative results were obtained for one adaptive compressed cache and static compressed caches with sizes ranging from 512Kb to 8Mb.

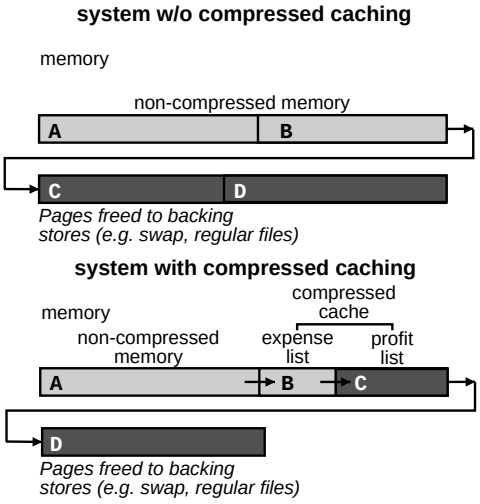

Figure 4: Lists in the compressed cache.

In terms of how the compressed cache adapts its size to system behavior, we begin with the assumption that compressed caching is useful to the system unless our online

analysis shows otherwise (i.e., first it does not have limitations to its growth). The analysis that detects when 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. In general, if compressed pages from the expense list (i.e. more recently used than pages on profit list) are reclaimed to the non-compressed memory, our policy takes this fact 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 (less recently used than the ones on expense list) indicate that compressed caching is beneficial to the system, since these pages would have to be read from disk if the system had no compressed caching at all. In more details, when the second consecutive page is read from the expense list, we lock the compressed cache growth, and if a third consecutive page is read from this list, we try to shrink the compressed page or free a compressed page. At this point, we reset the heuristic (i.e., even if a fourth consecutive page is requested from the expense list, it is taken as the first one for the heuristic). If any page is read from the profit list, we unlock the compressed cache growth and reset the heuristic as well. Details of this heuristic is found in Figure 5 below.

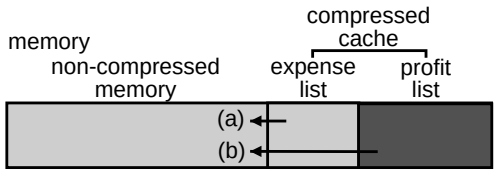

Figure 5: Adaptability heuristic. **(a) A page is read from the expense list.** If it is the second consecutive page, lock compressed cache growth. If third consecutive page, *(i)* Try to shrink the compressed cache relocating fragments, and if unable, *(ii)* free a compressed page. For cases where **(b) a page is read from the profit list,** unlock compressed cache growth (if locked).

From our experiments, we verified that our simple 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 better results than any static compressed cache. See Figure 3 for results of a comparison between a static and an adaptive compressed cache, relative to a kernel without compressed caching.

4. Experimental Results

In this section, we name the test suite and describe 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 is available on our Web site [10]. The results presented in this paper were run with the 0.24pre5 version of our code.

4.1. Test Suite

Our test suite is composed of the following tests: Linux Kernel 2.4.18 [5] Compilation, MUMmer 1.0 [14] (scientific program to align genome), Open Source Database Benchmark (OSDB) 0.14 [15] (database operation benchmark), httperf 0.8 [13] (web server benchmark), Matlab 6.0 [12] (mathematical tool), Sort from GNU textutils 2.0 [18] (sorting program) e PostMark 1.4 [16] (file system benchmark for small files). A detailed test suite description, the used tests and their input data are available on our Web site [10].

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 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 Algorithm

We provided support for the LZO [11] compression algorithm in our implementation. LZO is a fast Lempel-Ziv [21, 22] 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 `j4`), the MUMmer, the OSDB, 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. Each compressed cache kernel has a different configuration (different adaptability policy, number of pages in a cell, or type of pages it stores). The only two tests for which only the data of the two most important kernels were collected are httperf and Matlab, because they have not been used as basis for our design decisions.

test	memory	w/o CC	reference	sluggish	aggressive	onlyswap	cell1	cell4
	Mb	seconds	gain (%)	gain (%)	gain (%)	gain (%)	gain (%)	gain (%)
	18	467.8	21.73	18.07	21.46	-4.65	19.06	-0.55
	21	326.68	8.89	8.20	7.97	-1.01	7.33	1.64
	24	289.05	0.20	-0.35	0.66	-0.69	0.49	-1.56
kernel (-j1)	27	280.45	0.11	-0.64	0.17	-0.31	-0.19	-0.60
	30	278.33	-0.23	0.11	0.03	0.46	0.35	0.65
	48	274	0.18	0.09	0.28	0.47	0.26	0.34
	768	271.17	-0.27	-	-	-	-	-
	18	1002.62	33.13	11.68	31.76	1.60	28.86	-4.97
	21	608.98	33.84	21.99	30.40	-4.12	25.19	-4.41
	24	395.05	18.78	12.55	19.38	1.22	15.59	0.43
kernel (-j2)	27	313.8	5.37	6.80	6.92	0.00	6.32	-0.53
	30	283.7	1.12	0.92	0.24	-0.00	1.38	-0.23
	48	272.3	0.19	0.37	0.28	0.51	0.31	0.52
	768	269.76	-0.01	-	-	-	-	-
	18	1826.14	14.98	11.84	9.77	-0.52	13.31	-9.91
	21	1067.47	15.62	8.38	-1.50	-5.41	12.31	-11.72
	24	826.44	31.85	-2.16	20.60	-0.52	28.78	-7.65
kernel (-j4)	27	654.83	34.72	7.90	29.08	3.86	33.09	2.01
	30	489.67	26.45	13.45	25.47	0.84	26.48	5.89
	48	274.95	-0.39	-0.84	-0.64	-0.16	-0.48	-0.66
	768	271.23	0.28	-	-	-	-	-
	330	143.5	16.09	16.47	-120.36	-51.03	37.08	-29.84
	340	115.21	20.74	21.95	14.47	22.18	38.64	13.49
	360	82.86	26.25	26.64	24.04	24.33	24.13	25.67
	380	81.21	16.71	15.66	13.98	12.26	38.75	19.10
MUMmer	400	80.55	23.02	23.36	20.21	21.44	39.85	20.14
	420	58.51	15.11	14.19	14.10	16.34	14.80	10.48
	500	45.35	-0.22	-0.20	0.02	0.02	0.02	-0.07
	768	44.7	-0.09	-	-	-	-	-
	24	1242.4	30.70	31.59	31.91	-1.27	1.05	-7.69
OSDB	48	758.97	-0.07	-0.08	-0.63	0.75	-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	cell1	cell4
	Mb	reqs/sec	gain (%)	gain (%)	gain (%)	gain (%)	gain (%)	gain (%)
	24	38.5	171.38	-	-	-	-	-
	32	117.7	153.40	-	-	-	-	-
	36	1529.1	14.10	-	-	-	-	-
httperf	40	1849	1.40	-	-	-	-	-
	48	1646.1	14.95	-	-	-	-	-
	64	1819	3.20	-	-	-	-	-
	768	1894.1	-0.25	-	-	-	-	-

Table 1: Table with some of 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. *Onlyswap* has a compressed cache that stores only pages backed by swap. *Sluggish*, and *aggressive* have different adaptability policies. At last, *cell1* and *cell4* have cells composed of only one and 4 pages, respectively.

The *reference* column corresponds to the compressed caching kernel that achieved the best results in our experiments. It uses LZO compression algorithm, cells composed of two memory pages and the adaptability policy described in Section 3. (See Table 2 for some compression statistics.) 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. See Table 1 caption for details about other columns in that table.

	LZO
compression time	0.09 ms
decompression time	0.04 ms
compression ratio (kernel)	39.4%
compression ratio (mummer)	35.5%
compression ratio (osdb)	64.5%

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

We chose different amounts of available memory to the system in order to verify compressed caching varying from almost no memory pressure to high memory pressure. We also evaluated the overhead under no memory pressure at all. In order to do so, we ran 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 `j1` 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. 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 do not perform well in 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. In this case, *cell1* speeds up its performance by only 1.05% and *reference* by 30.70%.

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 only 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 (sort - 40.9% and postmark – 49.4%) when a compressed cache without the policy to disable clean pages compression is used. In the PostMark case, only 0.06% of all pages in the compressed cache are read back by the system, besides the compression ratio of 99% (almost all pages are incompressible). With our policy that disables clean pages compression, the slowdown which is noticed when 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. 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 applica-

tions 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 now have more saved faults. The Linux kernel responsiveness benchmark, namely contest 0.51 [9] has this behavior.

5. Related Work

In this section, we compare briefly our work with those ones that are closer to ours. We tried to compare our implementation with previous ones, but only the implementation 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). Neither Douglis' nor Cervera et al's implementations made their test suite available.

The first implementation of compressed cache was performed by Douglis [3] in 1993, achieving inconclusive results: speedups for some applications (up to 62.3%) and slowdowns for others (up to 36.4%). Kaplan [4], 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 had presented, 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). The second reason presented by Douglis for his inconclusive results was locality. In our implementation, any application that has many faults on pages that would be directly accessible without compressed cache and does not have enough benefits 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 also mentioned do not hinder compressed caching from providing improvements due to the high transfer rates nowadays. 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.

Kaplan [4] 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. 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.

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, no overhead at all is expected if no memory pressure is 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 considered a number of overhead concerns, particularly under memory pressure. Under this scenario, we observed that our simple 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. 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 in 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

Prof. Fabio Kon (IME-USP) 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. This research was supported by FAPESP through grant 01/01432-4, and CNPq through grant 465901.

References

[1] 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.

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

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

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

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

[6] 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.

[7] 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.

[8] 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.

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

[10] Compressed caching. <http://linuxcompressed.sourceforge.net/>.

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

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

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

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

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

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

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

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

[19] 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.

[20] 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.

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

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