Improving Application Performance through Swap Compression *

R. Cervera T. Cortes Y. Becerra

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya - Barcelona
http://www.ac.upc.es/hpc
{rcervera,toni,yolandab}@ac.upc.es

Abstract

The performance of large applications tends to be poor due to the high overhead added by the swapping mechanism. The same problem may be found in highly-loaded multi-programmed systems where many of the running applications have to use the swap space in order to be able to execute at the same time. Furthermore, those large applications might not be able to run on laptop or home computers as their resources are usually smaller than the ones found in an office system. In this paper, we present a solution to both problems that we have implemented in the Linux kernel. The idea consists of compressing the swapped pages and keeping them in a swap cache whenever possible. We have tested this new mechanism with a set of real applications obtaining a significant performance improvement.

1 Introduction

There are many applications that use large amounts of memory. These large applications take advantage of the swapping mechanism to run on the system as the available physical memory is not enough for them to run [12, 10]. The same problem appears when we try to run, on a laptop, the same applications we run on a desktop computer. These applications will relay on the swapping mechanism as laptop computers usually have less physical memory than desktop ones. Finally, multi-user environments tend to be very loaded and their applications have to swap out part of their memory so that all applications can run concurrently [16]. In all these cases, the performance of the applications is much lower than the one they would achieve if no swapping was needed. This happens because the swapping mechanism has to access the disk to keep the pages that do not fit in memory. It is clear that these applications, and the whole system, would benefit from a faster swapping system.

If we examine the same problem from a different point of view, we observe that increasing the number of pages that fit in the swap space without increasing the number of blocks in the swap partition would also be quite beneficial. We could run the same applications on a laptop than on a desktop system. Remember that laptops also have smaller disks if compared to desktop ones. This increase in swap space would also help multi-user systems to avoid getting out of memory. Finally, out-of-core applications could be programmed more easily as the global-memory restriction would not be so important.

Now a days it is quite normal to continue the office work at home. This usually means the use of large applications on a Linux box. These large applications fit well in the office machines but are too large to run efficiently on a smaller Linux box. In these cases, a fast swapping mechanism would be very beneficial as those applications would run faster and working at home would be less "painful". Furthermore, increasing the swap space at no cost would allow these kind of users to run applications that would normally not fit in their home machines.

These performance and space problems have motivated this work and its objectives. The first, and most important, objective is to speedup the swap mechanism. This will increase the performance of the applications that, for whatever reason, have to keep part of their memory in the swap space. It is also an objective of this paper to increase the size of the memory offered to the applications without increasing the number of disk blocks in the swap partition. It is important to notice that should these two objectives be in conflict, we will favor performance over capacity. Finally, we want to achieve both

*This work has been supported by the Spanish Ministry of Education (CICYT) under the TIC-95-0429 and the TIC-94-0439 contracts.

improvements with the minimum number of changes in the original Linux kernel.

The main idea used to accomplish both objectives consists of compressing the pages that have to be swapped out. This will increase the number of pages that can be placed in the swap partition. Furthermore, it will also allow us to build a cache of compressed pages that will decrease the number of times the system has to access the swap device. It is important to notice that previous studies show that good compression ratios can be achieved when compressing memory pages [7]. The idea we present in this paper is similar, in essence, to the one proposed by Douglis [4], but some improvements and modifications have been done (see Section 5). We believe that now is a good time to reevaluate the results obtained in this previous work as the technology has improved significantly which means that compressing and decompressing pages can be done much more efficiently.

This paper is divided into 6 sections. In Section 2, we describe the concepts and ideas in which this work has been based. In this section, we also present some preliminary results that will lead the final design. Section 3 gives a detailed overview of the way the mechanism works. Section 4 presents the benchmarks used and the results obtained while running them on our system. In Section 5, we present the most significant work already done in the area. Finally, Section 6 presents the main conclusions that can be extracted from this paper.

2 General Ideas & Frist Results

2.1 Caching

It has already been proved that caching is a good way to increase the performance of disk operations [13]. In our scenario, a cache for swapped pages should also increase the swapping performance if a few problems can be solved. One such cache would decrease the number of disk reads as some of the requested pages might be found in the cache. Swapping out pages could also take advantage of the cache as a swapped-out page might be freed before reaching the disk. Furthermore, if the pages have to go to the disk, the system could write many of these pages together in a single request. If we can write all of them sequentially in the disk, we will only have to pay the seek/search latency once per write instead of once per page.

Before we continue, it is a good time to go though some terminology that will be helpful throughput the rest of the paper.

Page: The virtual memory of applications is divided into portions of 4Kbytes. Each of these portions is know as a page.

Buffer: A buffer or cache buffer is a portion of 4Kbytes of memory where pages are stored before they are sent to the disk.

Disk block: This term refers the disk portion where the information of a buffer is stored. This means that disk blocks will also be 4Kbytes in size. We should take in mind that this term does not refer to sectors nor file-system blocks.

2.2 Compressing Cached Pages

Adding a cache to the swapping mechanism means that some memory available for processes is now taken away for the cache. This means that the applications will have less memory to work with. If nothing else is done, we have only taken some fast memory from the applications to offer the same amount of memory but somewhat slower. This does not seem to be the solution to increase the performance of the applications. The ideal solution would be to take some fast memory from the users to offer them a somewhat slower but 2 or 3 times larger one. Of course, this new memory has to be faster than the disk. This would reduce the number of times the system has to access the disk for paging reasons. This can be achieved by compressing the swapped pages. In a compressed cache, the system can keep more pages than the ones taken from the applications.

Whenever a page is swapped out, the system compresses it before storing it in the cache. On the other hand, when the swap module requests a page, the system gets it either from the cache or the disk and decompresses it before handling it to the swap module. Figure 1 shows the first version of the path proposed for swapping in/out pages.

2.3 Batching Multiple Pages Together

A second advantage offered by the cache is the possibility of batching multiple pages together to write them contiguously to disk. This idea was first proposed in

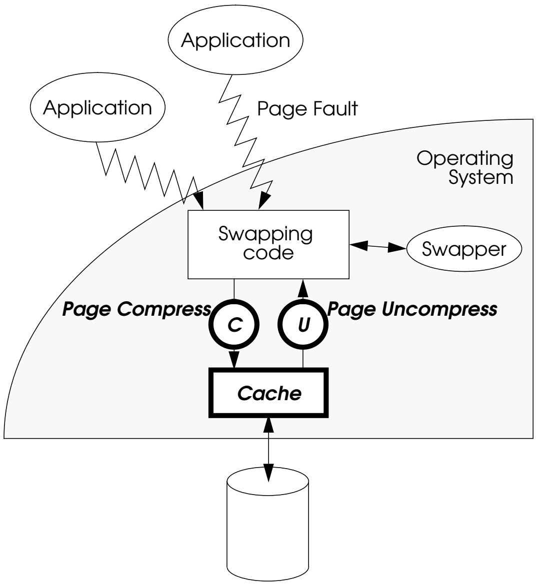

Figure 1: Conceptual vision of the compression and cache mechanism.

the VAX/VMS operating system [9]. This can be easily implemented as the system does not need to decide the physical location of a set of pages until they are really sent to the disk. Furthermore, as the pages are compressed, many more pages are written in a single disk write, thus decreasing the time spent on disk accesses.

2.4 Read/Write Path

Using all the proposed ideas, we built a preliminary prototype and we performed some measures and statistics. One of the most interesting results we obtained was the distribution of the two possible read-hit types: *read hit due write* and *read hits due write*.

Read hit due write: this kind of hits appear when the page is in the cache because it has recently been swapped out but has not yet been discarded. This means that the page is requested short after it was swapped out.

Read hits due read: this occurs when the page just requested is in a buffer that has recently been fetched from the disk. This means that another page, in the same disk block, has also been recently requested.

While examining both kind of hits, we detected that most of them were *hits due write*. This happens because the order in which pages are swapped out is not the same as the order in which they are swapped in. This led us to study the idea of not placing read buffers into the cache. This would allow recently written buffers to stay longer in the cache which might increase the hit ratio. Furthermore, this will also increase the write performance as less blocks will have to be sent to the disk.

In order to examine the effect of not placing read buffers in the cache, we implemented two versions of the preliminary prototype. A first one where the read buffers were placed in the cache and a second one where they were not. After running a set of benchmarks in both prototypes, we observed that the difference in the number of hits obtained by both systems was quite similar in most cases [3]. Furthermore, we also observed that the number of disk writes performed when reads do not interfere the cache is much lower than when reads are placed in the cache. This should increase the performance of the system as less writes are done and a similar number of reads are needed (similar read hit ratio).

Not placing read buffers in the cache has another interesting side effect. As reads do not need to make room in the cache, they will never have to perform a write operation to clean a dirty buffer. This will avoid many disk accesses while swapping in pages.

After this modification, the read disk blocks will not be placed into the cache. This does not mean that swapping-in operations will not take advantage of the cache. They will first try to find the page in the cache as it might have recently been written (*read hit due write*). If it is not in the cache, then the system will read the page, decompress it and forget about the rest of pages stored in the same disk block. Figure 2 shows the new path for swapping pages in and out.

Finally, another important side effect of not caching read requests is a simplification on the code. We will not get into many details now, but it is clear that a sapping-in operation will only have to search the page in the cache or to read it from the disk. It will not have to worry about cleaning buffers from the cache and it will also avoid most of the locking problems.

3 Prototype Description

In this section, we will describe the most important operations, policies and algorithm used in the final prototype. We will start describing the data structures used

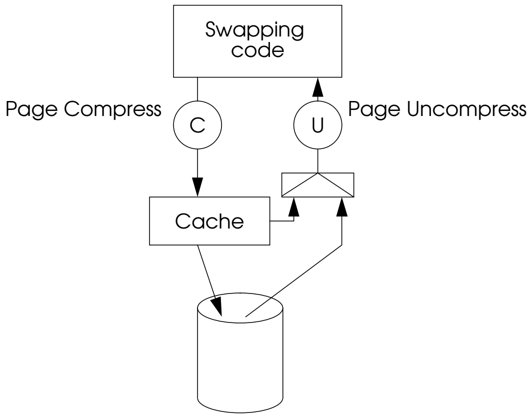

Figure 2: New swapping path where swapped-in pages are not kept in the cache.

and then, the four main operations will be explained with some detail. We have to keep in mind that only the main ideas are described and that technical issues such as locking or very infrequent situations are not presented in the paper.

Regarding some implementation details, the prototype has been built in the Linux operating system (kernel version 2.0.34) [2]. The compression algorithm chosen has been *lzo* [11] which is based on the Ziv-Lempel data compressor [17]. This algorithm was chosen as it obtained a good ratio between speed and compression. On one hand, it achieved compression ratios better than 50% in most of the experiments (Table 1). On the other hand, the average time needed to compress a 4Kbyte page is about 300 microseconds while the one needed to decompress a buffer is only about 50 microseconds [1].

3.1 Data Structures

In order to implement this mechanism we have added some data structures to the original Linux kernel. In this section, we will describe each of these structures in some detail. A general picture with all the structures and most of the fields is presented in Figure 3.

- `virtual_swap_info`: this structure keeps the information of all the compressed pages. The size of this array is MAX_VPAGES, which is the maximum number of compressed pages that our system will be able to handle. The size of this array can be modified to suit the needs of each system as explained in Section 3.6. In each entry of this array we have the following information:

    `physical_offset`: it indicates the disk block where the compressed page is stored.

    `buffer_offset`: it is the field used to mark the position where this compressed page is kept. As we store more than one compressed pages per disk block, we need to know at which byte does the page start.

    `compressed_size`: as can be guessed from its name, this field is used to keep the size of the swapped page once it has been compressed.

    `rw_remaining`: it is a counter of the number of pending read or write operations for this page. We need this information not to free a page while still being used.

- `swap_map`: this structure was already used in the original Linux kernel. We have only modified its size as we need an entry for each one in the `virtual_swap_info` array. Its function is to keep the number of processes that have this page mapped in their address space.

- `physical_swap`: for each disk block, we need to know the number of compressed pages kept in it.

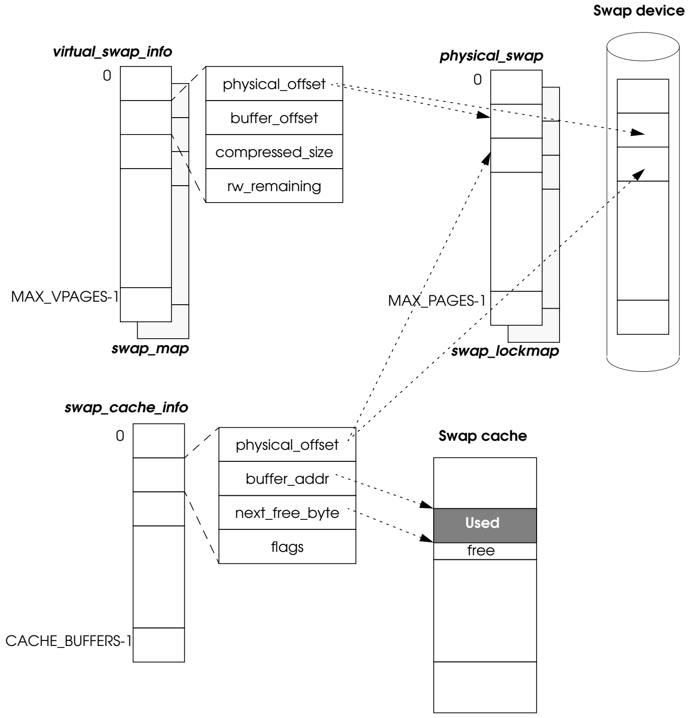

Figure 3: Data structures needed for the compressed swap.

[1] Measured on a Pentium II at 350MHz

This table is responsible for maintaining this information.

- `swap_lockmap`: there are situation where we need to perform atomic operations on the disk block. To ensure that no other process will work with a given block, we use this data structure already implemented in the original Linux kernel. It is a bitmap where each bit tells whether the given disk block is being used in exclusive mode or not.
- `swap_cache_info`: this structure is used to keep all the information needed to maintain the compressed cache. It has as many entries as buffers in the cache (CACHE_BUFFERS).

  `physical_offset`: it indicates the disk block assigned to this buffer.

  `buffer_addr`: it points to the cache buffer where the compressed pages are really stored. We need this pointer as all buffers are not necessarily contiguous. This is because we cannot allocate as many buffers as needed in a single call (Linux implementation issues).

  `next_free_byte`: it keeps the first free byte. This is the position where the next compressed page inserted in this buffer will be placed.

  `flags`: cache buffers need some flags such as the dirty bit.
- `swap_cache`: this structure is just a set of buffers (not necessarily contiguous) that are used to keep the compressed pages before they are sent to the disk.
- `Swap device`: finally, this is the disk partition where the compressed pages are finally stored.

3.2 Getting Free Space (`get_swap_page`)

In the original kernel, this function returned the disk offset where the page would be stored but this offset was used as an identifier for the swapped page. Only the swap code used it to access the disk. As we need to know the size of the compressed page before assigning it to a disk block (it has to fit in it), we will return an index to the `virtual_swap_info` table. To the rest of the kernel, this function behaves as always and the system believes that the swap partition is a larger one.

To return this index, the system searches for a free entry in the `virtual_swap_info` array. A given entry is free when no process is using it (`swap_map[i] == 0`) and when there are no operations remaining to be done on this page (`virtual_swap_info[i].rw_remaining == 0`).

Another important issues is that the system cannot return an index unless it is sure that there will be enough disk space to keep the page. For this reason, the system will always assume the worst case. Until it knows the real compressed size, the system will assume that the page needs a full disk block to be stored. As soon as the system knows its real size, it will update this information.

3.3 Freeing a Page (`swap_free`)

Freeing a page consists of decrementing the number of processes that are using the page (`swap_map[i]--`). Whenever this becomes zero, the system has to free this space from the cache buffer or the disk block. This is done by decrementing the number of compressed pages in its disk block or cache buffer (`physical_swap[i]--`). Should this page be the last one in the buffer, the whole buffer should also be freed to be used by other pages in the future.

The above scenario is the best possible case. For mutual exclusion reasons, the order in which the operations are done can be broken and a page might be freed while a write operation is still pending. In this case, the page is marked to be freed as soon as the pending operations are finished. We will see this while describing the swapping in and out operations.

Another important issue related to freeing a page is the recompactation of blocks and buffers. If the page that is being freed was stored in the middle of a block or buffer, we could think of reallocating all pages together to merge this new space with the free space already remaining in the block or buffer. Doing this on disk blocks is completely out of the question as it would mean reading the disk (too much overhead). Recompacting cache buffers is a feasible task but we have seen that it does not increase the performance of the system and makes the code more complex [3]. For these reasons, we will never reuse the space of a freed page until all the compressed pages in a block or buffer have been freed.

3.4 Swapping Out (`rw_swap_page`)

Once the system wants to swap out a page, it compresses the page and tries to write it into the cache. Performing this operation, we might find that there is not enough free space to cache this new page. This means that all buffers are dirty (and have not been sent to the disk) and that all of them have less free space than the size of the compressed page. It is important to notice that the system will never split a compressed page among several buffers. When the system runs out of free space in the compressed cache, it performs a cleaning operation. Once it is done, at least one buffer will not be dirty and the system will be able to use it to put the new compressed page.

As we mentioned when describing the free operation, there are cases where a page could not be freed because there was a write operation still pending. If this is the last pending write operation and the page is marked as to be freed, the system will free the page after the write has been finished. The same steps as in the original free operation are taken.

Cleaning Mechanism

To clean the cache we need to send one or more buffers to the disk. This will allow the system to reuse them as an up-to-date copy of the data will be kept in the swap device. The intuitive idea of cleaning the cache consists of sending to the disk all buffers when no free space is left. As buffers do not usually get completely filled with compressed pages, we have modified the concept of full buffer as follows:

Full buffers are those ones that have less free space than the average size of the last 100 compressed pages.

Using this new concept, whenever a page does not fit in the cache, all *full buffers* will be sent to the disk in a single operation where all of them are written contiguously on the disk. This will increase the performance of the write operations significantly.

Should the system need to perform a clean operation when there are no *full buffers*, the buffer with more data will be sent to disk.

As we do not want to wait until no free space is left on the cache to clean it, whenever it has a given percentage of its buffers *full*, the system writes them to the disk. This percentage can be adjusted to the needs of the system as will be seen in a later section. This operation is currently done in a synchronous way but we are working to make it asynchronous.

An important detail is the addition of a flag that tells whether there is a cleaning operation already running. If this flag is on, a second concurrent cleaning will not be done as only one is really needed.

3.5 Swapping In (`rw_swap_page`)

This is the simplest operation. Whenever a pages is requested, the system searches for it in the cache. If the page is found in the cache, the system decompresses it and places the result on the user address space. Otherwise, if the page is in the disk, the disk block is read and the page is decompressed as in the previous case. As read disk blocks are not placed in the cache, if another page from the same buffer is requested, a new disk read will be needed (remember that *hits_do_read* are not very frequent).

3.6 Driver to Modify the Parameters

To simplify the task of setting the parameters for the compressed swap we have also implemented a diver that allows the superuser to modify the following parameters when the swap is off.

Virtual space size: maximum number of compressed pages that the system will be able to handle (`MAX_VPAGES`).

Cache size: the number of Kbytes used for caching.

Cleaning threshold: the percentage of full buffers the system needs to find to perform a cleaning operation.

4 Experimental Evaluation

4.1 Methodology

All the results presented in this paper have been measured on a Pentium II running at 350MHz. The amount of physical memory was 64 Mbytes, and the size of the swap partition was 128Mbytes. This partition was located on a Ultra-SCSI hard disk.

All the measures presented are the average of, at least, 10 executions, in single-user mode, where the best and worst ones have been discarded.

4.2 Benchmark Description

To measure the performance of this proposal, we need to see the effect it has on a set of benchmarks. Before getting into a detailed description of the benchmarks, we would like to describe the three characteristics a benchmark can have that may have a higher effect on the behavior of the proposed system.

Concurrency. It is important to see that the number of processes in the benchmark will affect the behavior of the system. If only one process is running in the system, the application that is swapping out pages will not have to wait for another application that may have locked some of the resources it needs. No other application will try to swap in/out pages.

I/O. Another benchmark parameter that will affect the system is the amount of file-system I/O performed by the benchmark. This I/O may conflict with the one performed by the paging system because both are done in the same disk (although in different partitions).

Compression ratio. [2] Finally, the compression ratio may affect the system in two ways. First, the better pages compress, the larger the final size of the swap area will be. Second, if pages have a good compression ratio, the number of pages that can be kept in the cache will be higher. Thus the number of disk accesses should be lower than with bad compression ratios.

Once described the most important characteristics, we will describe the benchmarks used.

- `fft.` It executes a fast Fourier transformation with a 2048x2048 matrix. The values of the elements in the matrix are set randomly.
- `fft x10.` This benchmark is very similar to the previous one but 10 ffts are executed concurrently and the size of the matrixes is decreased to 512x512 elements.
- `sort.` In this benchmark, we perform an in-memory sort of a text file. The input file is build by appending the `/usr/dict/word` file many times and then unsorting it as much as possible.
- `sort x6.` In this benchmark 6 sorts are executed concurrently. The file to be sorted is built as the previous one but 5 times smaller to limit the execution time of the benchmark.
- `simulator.` A simulator of a network of disks currently being used in our research group. It is an event-based simulator that uses large amounts of memory. This memory compresses very well as many of the events in the queues have similar information. Furthermore, the memory library used, does not free the allocated memory after a `free`, the library keeps the memory block in a hash queue for further use. This ”freed” memory is also easy to compress. Although the compression ratio of this application seems to be unrealistic, there are other applications in a typical Unix system that achieve compression ratios better than 10% such as `awk` [7].
- `simulator x5.` Five concurrent executions of the simulator but with a smaller input.
- `xanim.` A visualization of a video file in `avi` format. This video is dithered using the Floyd-Steinberg algorithm. This means that the file has to be decompressed in memory to do the dithering before it is visualized. This benchmark has been run under the X-Windows system as it needed to perform graphic I/O.
- `xanim x4.` Four concurrent executions of the previous xanim benchmark.

Table 1 summarizes the characteristics of each benchmark according to the parameters described in the first part of this subsection.

4.3 Performance Results

As this paper just tries to show that this mechanism is useful to increase the performance of large applications we will only present a selection of all the possible experiments. We will show the effect of the cache size given a cleaning threshold and the effect of this threshold given

[2] $compression\_ratio = compressed\_size/page\_size$. This means that high percentages denote bad compression

	Concurrent processes	File I/O	Compress ratio
fft	1	none	64.9%
fft x10	10	none	61.2%
sort	1	start/end	46.5%
sort x6	6	start/end	51.3%
simulator	1	start/end	6.7%
simulator x5	5	start/end	1.1%
xanim	1	first part	27.8%
xanim x4	4	first part	37.9%

Table 1: Benchmark characteristics.

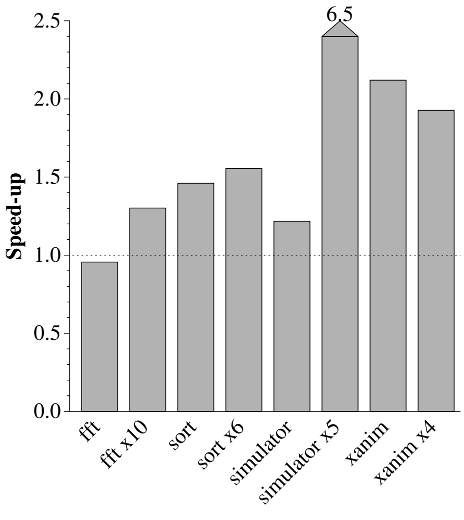

Figure 4: Effect of the compressed swap on several workloads.

a cache size. Both parameters should be tuned for each system depending on the hardware and expected load. Anyway, even if the best configuration is not chosen, most reasonable configurations will imply a significant performance improvement.

General Performance Results

In this first experiment, we have configured the compressed cache using what we though would be nice parameters. We have used a 1Mbyte cache and a cleaning threshold of 50% of the cache. Using these values, we executed all the benchmarks and computed the speedup obtained when compared to the original swapping mechanism. Figure 4 presents these results.

In this graph we can see that all benchmarks but one observe a speedup between 1.2 and 2.1. This means that these applications run, at least, a 20% faster than with the original swapping mechanism and there are even executions where the applications half their execution time. The two exceptions to this rule are `fft` and `simulator x5`.

The first one (`fft`) achieves a speedup of 0.96, which means that it runs slower than with the original system. This slowdown is due to two basic factors. The first one is that the compression ratio is not very good and most pages cannot be compressed less than 2048 bytes. This means that it is quite difficult to place more than one page per buffer or disk block. The second reason is that taking memory from the application for our data structures and cache buffers has a significant effect on the application. Without this memory, the working set of some parts does not fit in memory anymore and the application pages much more than with the original system.

The second exception to a reasonable speedup is the execution of 5 concurrent simulations (`simulator x5`). This benchmark achieves a speedup of 6.5. Such an impressive improvement is due to its incredible compression ratio. As pages compress so well, most swapped pages fit in the cache and nearly no disk access are needed.

These two exceptions will not be very frequent and we should expect a performance improvement between 20% to 100%, which is a significant gain.

Another unexpected result is the low speedup obtained by the `simulator` benchmark. As this benchmark compresses very well (6.7%), we expected to have a much more important speedup. The reason behind this behavior is the well behavior it has on the original system. As it swaps out many pages in very small periods of time, the original system can group them together before sending them to the disk and performs something similar to a batched write. For this reason, the gains we obtain by batching write operations together is also gained by the original system. This situation only happens in the original kernel when many pages are swapped out while writing the disk is busy. The kernel coalesces all these requests in a single one if contiguous. Anyway, this does not happen too often as we can see from the speedups obtained by the other benchmarks.

Cache-size Influence

The second experiment tried to study the influence of the cache size in the performance of the new mechanism. To do this experiment we have run all the benchmarks vary-

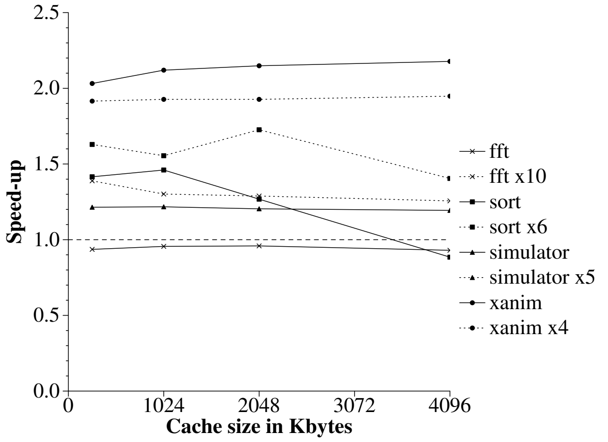

Figure 5: Influence of the cache size on the application performance.

ing the cache size between 256Kbytes to 4Mbytes and the cleaning threshold used in all these experiments was 50%. The obtained results are drawn in Figure 5. In this graph, we will not present the results of `simulator x5` as a curve with speedups of 6 would difficult the study of the graph.

The main observation is that there is nothing such as a perfect cache size for all benchmarks. It is clear that very large caches are no good as they take too many pages from the applications and they have to swap far too much.

Anyway, the important thing is that with reasonable cache sizes (around 1Mbyte) the performance of the applications is greatly improved due to the compressed cache.

Cleaning Threshold

The final parameter is the cleaning threshold. This value defines the percentage of buffers that have to be *full* before a cleaning operation is started. To study the influence this parameter has, we have set the cache size to 256Mbytes and we have varied the threshold from 1% to 70%. The results obtained by all the benchmarks are presented in Figure 6. The performance of `simulator x5` is also not included in this graph for the same reason as in the previous subsection.

We can observe that this parameter cannot be too small. In this case, many small write operations are performed and the seek and search actions become an important overhead in these write operations. When the threshold grows, the performance of the applications tends to increase as the disk latency becomes less important.

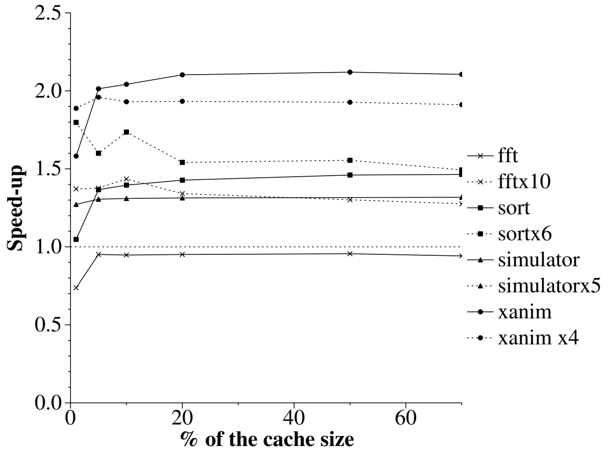

Figure 6: Influence of the cleaning threshold on the application performance.

For very large values, the behavior of the system depends on whether the benchmark is a mono or multi-process one. In mono-process benchmarks, when the cache is being cleaned (synchronous cleaning), no other process accesses the disk and the benefits of large writes continue to be the a good issue. On the other hand, in multi-process benchmarks, while a process is cleaning the cache, another may want to swap in or out a page. If the cleaning operation is too long, the read/write operation has to wait for a long time and the performance of this process is also affected in a negative way.

The benchmark `sort x6` has a very unpredictable behavior. It does not follow any clear pattern. The reason behind this behavior is the large amount of I/O it performs. If it tries to write while the system is cleaning the cache, the performance is greatly affected. There is no way to avoid these collisions, but still the results are good enough.

It seems that the best value for this threshold is between 10% and 20%.

4.4 Increase of the Swap Space

So far, we have only seen the performance benefits of compressing the swap area. As we mentioned in the introduction, this was the main objective of the project. Anyway, we also had a second objective that consisted on increasing the number of pages that could be placed in the swap area. A study of this objective is presented

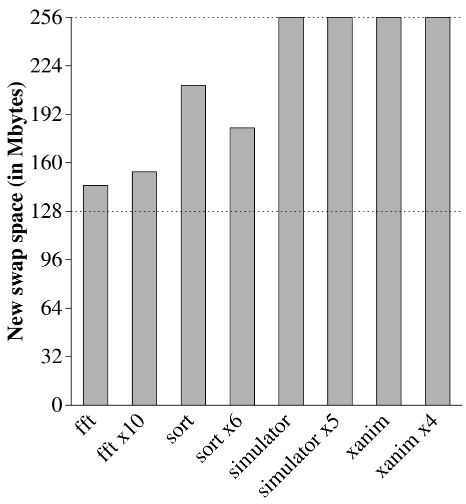

Figure 7: Size of the swap area that would be needed to have the same capacity as our 128Mbytes compressed swap.

in this subsection.

Although the number of pages that fit in the swap partition depends on the compression ratio, it is not the only important factor. The fragmentation found inside the buffers will also be an important parameter in the final size of the swapping space. A system that leaves large unused portions in the buffers will not be able to place many more pages than the original system in the swap partition.

Figure 7 presents the size of the swap area that we would obtain if we could fill it with pages following the same compression ratio and the same fragmentation as the ones obtained in the benchmarks. In the figure, we can see that in most cases the system increases the size of the swap partition more than a 50%.

We have limited the gain to 256 Mbytes as we have configured the size of the `virtual_swap_info` table to double the physical swap space. If a greater array were used, a larger swap space would have obtained with benchmarks such as `simulator` and `xanim`.

5 Related Work

Not much research has been done in the area of compressing the swap space. The *compression cache* proposed by Fred Douglis [4] is very similar, in essence, to our work, but some important differences can be found. In that work, the swap pages are also compressed and kept in a cache to increase both the size of the virtual memory and the performance of the applications that have to swap. One big difference between our work and the one done by Douglis is that the results we present are not so dependent on the compression ratio as they were. In the previous work, no performance gains were obtained with compression ratios worse than 30% while we obtain nice performance improvements with even a compression ratio of 62%. This might be either due to design issues or due to the improvements in the technology (compressing is much faster now). It is also important that the previous work lacked a study on the kinds of read hits obtained in the cache. This study has led us to significant design modifications such as having two different paths: one for swapping in and one for swapping out. As we have shown in this paper, this distinction has obtained significant performance benefits. Finally, all their benchmarks were single process while we believe that multi-process benchmarks have also to be studied.

If we examine our work in a more general way we can divide it in two basic issues: increasing the size of the memory and reducing the average time needed to swap in/out a page. Let's discus what has been done in both fields.

Following the idea of increasing the size of the memory, there are some commercial products that compress the physical memory. With these software mechanisms the applications believe that the system has a larger amount of physical memory. Anyway, the achievements obtained by such systems are not clear [8, 14]. The same idea has also been done in hardware with much better performance gains [6].

There have also been many proposals to decrease the number of disk accesses for swapping issues. For instance, some work has been devoted to minimize the number of pages that have to be swapped out. If the contents of a page is irrelevant to the application execution, this page does not need to be kept in the swap [15, 5]. In the same line, software has been developed to study the utilization of the pages and thus improve the programs and reduce the number of pages swapped in/out [12]. There has also been some work that tried to group pages when swapped out so that larger writes were done [1].

Finally, the approach of compressing information before sending it to the disk is widely used in database environments and in some file systems.

6 Conclusions

In this paper, we have presented a way to implement a compressed-swap mechanism that achieves significant improvement in the performance of lager applications. Most of them achieve speedups between 1.2 and 2.1 and there are some special cases where this speedup is even much higher.

We have also shown that, although the configuration affects the performance, it is not difficult to find a reasonable set of values that work well with all applications.

Finally, this mechanism has been installed in some Linux boxes in our department and the users are quite happy with this new feature.

Acknowledgments

We would like to thank Professor J. Labarta and X. Martorell for all their help. Their interesting comments increased the quality of this work significantly. We are also grateful to E. Artiaga who helped us with some of the benchmarks used in while testing the proposed mechanism. Finally, we are also in debt to J. Fischer and E. Youngdale, which had the patience of guiding us into many important details of the SCSI driver for Linux.

References

[1] BLACK, D., CARTER, J., FEINBERG, G., MACDONALD, R., SCIVER, J. V., WANG, P., MANGALAT, S., AND SHEINBROOD, E. OSF/1 virtual memory improvements. In *Proccedings of the Mach Symposium* (November 1991), USENIX Association, pp. 87–103.

[2] CARD, R., DUMAS, E., AND MÉVEL, F. *Programming Linux 2.0*. Editions Eyrolles, 1997.

[3] CERVERA, R., AND CORTES, T. Swap comprimit: Disseny i implementació. Tech. Rep. UPC-DAC-1998-34, Universitat politècnica de Catalunya, Departament d'Arquitectura de Computadors, 1998.

[4] DOUGLIS, F. The compression cache: Using online compression to extend physical memory. In *Proceedings of the Winter Technical Conference* (January 1993), USENIX Association, pp. 519–529.

[5] HARTY, K., AND CHERITON, D. R. Application-controlled physical memory using external page-cache management. In *Proccedings of the V Architecture Support for Programming Laguages and Operating System* (October 1992).

[6] KJELSO, M., GOOCH, M., AND JONES, S. Design and performance of a main memory hardware data compressor. In *Proceedings of the 22nd Euromicro Conference* (September 1996), IEEE Computer Society Press, pp. 423–430.

[7] KJELSO, M., GOOCH, M., AND JONES, S. Empirical study of memory-data: Characteristics and compressibility. In *Proceedings IEE, Comput. Digit. Tech.* (January 1998), vol. 145, pp. 63–67.

[8] LEONARD, D. Magnaram 97 is no cure-all for the ram blues. C|net Special Report (http://www.cnet.com), October 1997.

[9] LEVY, H., AND LIPMAN, P. Virtual memory management in the VAX/VMS operating system. *IEEE Computer 15*, 3 (March 1982), 35–41.

[10] MOWRY, T. C., DEMKE, A. K., AND KRIEGER, O. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 2nd International Symposium on Operating System Design and Implementation* (1996), USENIX Association.

[11] OBERHUMER, M. F. Lzo v1.04. http://wildsau.idv.uni-linz.ac.at/mfx/, March 1998.

[12] ROBINSON, E. M., AND LEIS, E. L. Page utilization in fortran and C programs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* (July 1998), CSREA Press, pp. I:206–210.

[13] SMITH, A. J. Disc cache - miss ratio analisys and design considerations. *ACM transactions on Computer Systems 3*, 3 (August 1985), 161–203.

[14] STEERS, K. Tune up your memory. *PC World 24* (August 1997).

[15] SUBRAMANIAN, I. Managing discardable pages with and external pager. In *Proccedings of the 2nd Usenix Mach Symposium* (November 1991).

[16] VERGHESE, B. *Resource Management Issues for Shared-memory Multiprocessors*. PhD thesis, Sanford University, March 1998.

[17] Ziv, J., and Lempel, A. Compression of individual sequences via variable-rate coding. *Transactions on Information Theory 24* (September 1978), 530–536.