2008 14th IEEE International Conference on Parallel and Distributed Systems

Improving Web Server Performance Through Main Memory Compression

Vicenç Beltran, Jordi Torres and Eduard Ayguadé
Barcelona Supercomputing Center - Technical University of Catalonia
{vbeltran, torres, eduard}@ac.upc.edu

Abstract

Current web servers are highly multithreaded applications whose scalability benefits from the current multicore/multiprocessor trend. However, some workloads can not capitalize on this because their performance is limited by the available memory and/or the disk bandwidth, which prevents the server from taking advantage of the computing resources provided by the system. To solve this situation we propose the use of main memory compression techniques to increment the available memory and mitigate the disk bandwidth problem, allowing the web server to improve its use of CPU system resources.

In this paper we implement to the Linux OS a full SMP capable main memory compression subsystem to increase the performance of a web server running the SPECweb2005 benchmark. Although main memory compression is not a new technique per-se, its use in a multicore environment running heavily multithreaded applications like a web server introduces new challenges in the technique, such as scalability issues and the trade-off between the compressed memory size and the computational power required to achieve it. Finally, the evaluation of our implementaiton shows promising results such as a 30% web server throughput improvement and a 70% reduction in the disk bandwidth usage.

1. Introduction

Generally speaking, compressed memory systems are based on the reservation of some physical memory to store compressed data, virtually increasing the amount of memory available to the applications. This extra memory reduces the number of accesses to the disk and allows the execution of applications with larger working sets without trashing. However, the benefits of the compressed memory systems greatly depends on both the application access pattern and the data compression ratio, as well as, the ratio of compressed/uncompressed memory configured.

Previous work has exploited the compressed memory systems to accelerate the execution of single threaded applications with a large working set, exchanging high latency disk access for faster compressed memory access. This approach uses the idle times that this type of application usually spends accessing the disk to perform the decompression of the data requested. In contrast, we are interested in investigating the benefits that compressed memory systems can contribute to disk I/O bandwidth bound applications like a web server running the SPECweb Support workload. In this case, the problem is that the web application is bounded by the available I/O bandwidth of the disk. A compressed memory system can mitigate this problem by providing more available memory to cache disk content in memory, thus reducing the number of accesses to the disk and the effective disk I/O bandwidth needed. A major challenge with this approach is the large amount of CPU power needed to provide the adequate bandwidth between the non compressed memory and the compressed one and viceversa. However, this CPU power is now more easily available with the proliferation of multicore and multiprocessor systems which can be utilized for this purpose.

In summary, the focus of the paper is to improve the performance of a highly multithreaded web server running a disk-bounded web application. To this end, we have implemented the first full SMP capable Compressed Page Cache (CPC) in the Linux OS that make the most of multicore/multiprocessor architectures. To accomplish our objective we have solved two challenging scalability issues. Firstly we have implemented our CPC on a multiprocessor Linux system that can scale in a highly threaded environment like that provided by a web server running the SPECweb2005 benchmark and secondly, we have fructuously used a large fraction of the physical memory to store compressed data without running out of memory. As we will show, our novel CPC proposal is able to work with optimal performance when up to 85% of the memory is dedicated to store compressed data (in contrast to previous proposals that have been evaluated with a much smaller fraction of compressed memory, e.g. 10-20% in [19]).

1521-9097/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPADS.2008.15

2. Related Work

Web servers are a well studied subject in the literature; issues like performance [7], scalability [11], overload [12] or security [5] have been deeply discussed but, to the best of our knowlegde, the improvement of web server performance with memory compression techniques has not been studied before. The rationale behind using memory compression techniques to improve web servers' performance is based on the fact that the bottleneck of a web server for some workloads is the disk bandwidth and we can mitigate it with more memory at the expense of CPU cycles to perform the data compression. With the expansion of multicore and multiprocessor systems the CPU resources needed to make this technique feasible are currently available.

Our software main memory compression implement the compression techniques on top of commodity hardware without any special hardware support. In this section we review the most relevant works in this area. The first memory compression proposal, due to Wilson [21], intends to improve system performance reducing the latency associated with disk access. In [10] Douglis implements the first adaptive memory compression scheme in Spirit OS, based on a global LRU that can improve or decrease the performance of the system depending on the workload characteristics. Kaplan et al. [13] study the adaptive memory compression scheme proposed by Douglis through simulation and found that the proposed scheme has been partly at fault for some workloads. Kaplan also contributes the WK family of compression algorithms designed for in memory data representations rather than file data. Finally he proposes a method to determine how much memory should be compressed during a phase of program execution by performing an online cost/benefit analysis, based on recent program behavior statistics.

In [8] Cervera et al. implement in the Linux OS a compressed swapping mechanism to reduce the number of times the system has to access the swap device. Although the amount of compressed swap memory used was rather small, they observe a noticeable improvement of system performance. This is the first work that swaps out pages to the swap device in compressed form, virtually increasing its capacity. Freedman et al. [3] apply memory compression techniques to reduce the power consumption and to improve the speed of embedded systems. Their compressed cache implementation is based on a log-structured circular buffer that allows the compressed cache area to be dynamically resized. They estimate that compressed memory improves the disk acces in both power efficiency and speed by 1-2 orders of magnitude. In [17] Roy et al. also proposes using compressed memory in order to hide the large latencies associated with disk access. They claim that the optimal fraction of memory that should be reserved for compression lies at around 25% across a wide range of application types but they fail to provide a more general approach to set the memory compression size. In [9] Rodrigo reevaluates the use of adaptive compressed caching to improve the system performance. The main idea behind their proposal remains and it is to reduce the amount of disk accesses to improve the data access latency. Their contribution is a new adaptability policy that adjusts the compressed cache size on-the-fly based on the recent program behavior. They implement the compressed cache in the Linux kernel and it's the first to provide file backed memory compression as well as swap based memory compression. They use the WKdm specialized compression algorithm to compress swap based pages and the LZO generic algorithm to compress file based memory pages. Their implementation provides noticeable improvements for a wide range of workloads and minimum overhead for the rest. Tuduce [19] proposes a new heuristic to dynamically determine the compressed cache size with the objective of keeping all the application's working set in memory. Their results show increases in performance by a factor of 1.3 to 55 times in three single threaded applications. Finally, in [16] Nitin Gupta has ported Rodrigo's implementation of the compressed cache from kernel 2.4 to kernel 2.6 under the Google Summer of Code program for the OLPC project [15]. The work is based on the work and ideas of Kaplan, Rodrigo and Irina and the main objective is to increase the tiny memory available on the OLPC laptops.

None of the cited works has studied memory compression from the point of view of disk I/O bandwidth. We focus our discussion around the multicore and multiprocessors systems as today they are standard commodity hardware. To the best of our knowledge, our implementation is the first to fully take advantage of the new multicore and multiprocessor system'. Another remarkable characteristic is that it is highly scalable in the amount of RAM that can be used to store compressed data (up to 85% of the physical RAM). In this paper we do not compare our implementation with previous memory compression proposals (like [9] and [19]) because it would produce equivalent results for the workloads that have already been studied. Instead, we focused on the evaluation of a multithreaded web server with a disk bandwidth-bound workload on a multiprocessor environment, which is not supported by previous implementations of the main memory compression techniques described in the literature. We have augmented our compressed page cache, first introduced in [6], with an asynchronous implementation described in section 4.4. We have evaluated our compressed page cache with a complete scalability study in a quad-core PowerpPC970 server. We also analyze the tradeoff between the compressed memory size and the CPU power required to perform the compression and decompression tasks.

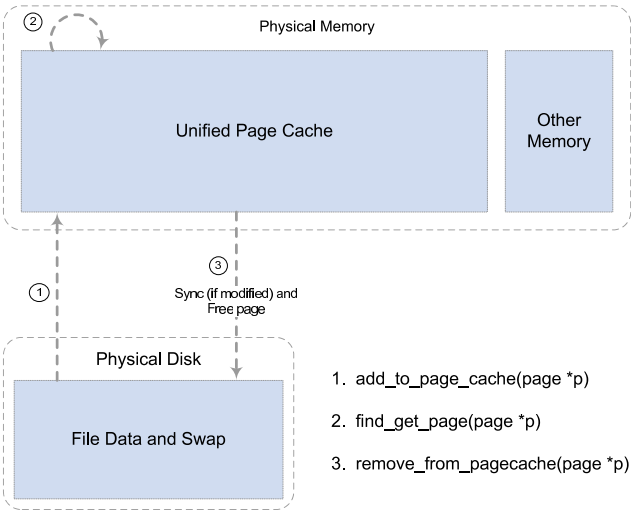

Figure 1: Linux Unified Page Cache Diagram

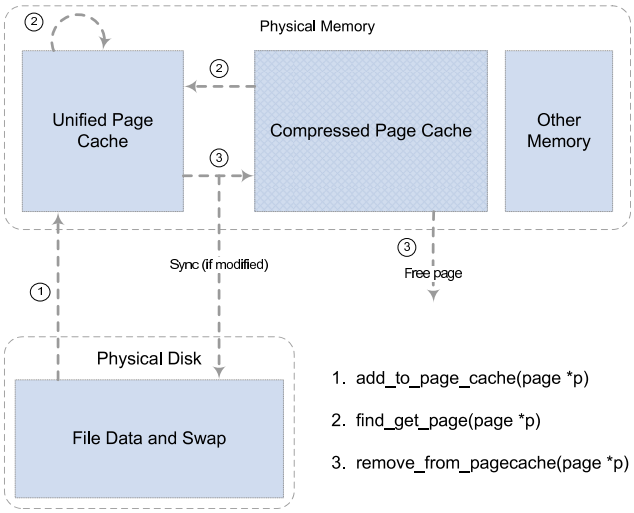

Figure 2: Compressed Page Cache Diagram

3. Compressed Page Cache Design

We have chosen the Linux operating system to implement our compressed memory subsystem. In the following section, we briefly describe the overall Linux memory management subsystem and the design goals of our compressed page cache (CPC). For more details about our CPC implementation refer to [6].

3.1. Design Goals and Implementation

The objective of our design is to extend the unified page cache of Linux depicted in Figure 1 and provide a high performance compressed page cache (CPC) that can fully exploit the power of current multiprocessor systems. We also want to be able to use a large fraction of the physical memory to store compressed data because our web server workload (SPECweb Support) has a huge working set, many times larger than the available memory. Another design objective is to minimize the number of changes made to the Linux kernel, avoiding the addition of complex algorithms or data structures. The Figure 2 shows the Linux unified page cache extended with our compressed page cache.

Linux memory management is developed around its core concept: the page frame. All the memory available on the system is divided in to page frames of the same size (usually 4Kb). The page frame is the smallest unit of work to manage the system memory. The content of a page frame changes dynamically depending on the needs of the system. Generally speaking, one page frame may contain three types of data: anonymous pages, file backed pages and private kernel pages. The anonymous pages contain data dynamically allocated from user space programs and can be swapped out under memory pressure if a swap device exists. File backed pages contain data that comes from filesystem I/O operations. Finally, private kernel pages are used and managed by the kernel and device driver code for private purposes and can not be swapped out. An example of this type of memory is the SLAB allocator, which provides memory for in-kernel use. The anonymous pages and the file backed pages form the unified page cache. The main data structure behind the unified page cache is a radix-tree that works as an efficient dictionary, mapping keys with page frames. All the I/O operations take place through this unified cache depicted in Figure 1. When a page fault occurs or an I/O operation is required, the kernel always checks the unified page cache (with a call to find_get_page()) to find the requested data. If the data is not in the page cache the kernel adds a new page frame to the page cache (with a call to add_to_page_cache()) and performs the required I/O operation so that the page cache is always up to date. All the pages of the unified page cache are linked together with a linked list to track their activity with a LRU like algorithm. When the system is under memory pressure, the kernel tries to free batches of pages from the tail of the LRU list (with a call to remove_from_pagecache()) until enough memory is available.

The main idea behind the CPC is to modify the current unified page cache depicted in Figure 1 to also contain compressed page frames. Each compressed page frame has an augmented struct page called struct cpage, which is dynamically created to manage its content. This struct cpage is an extension of the standard struct page but contains additional information about the location and size of the compressed page frame. We mark one unused bit of the flags field in order to identify the pages that are currently compressed.

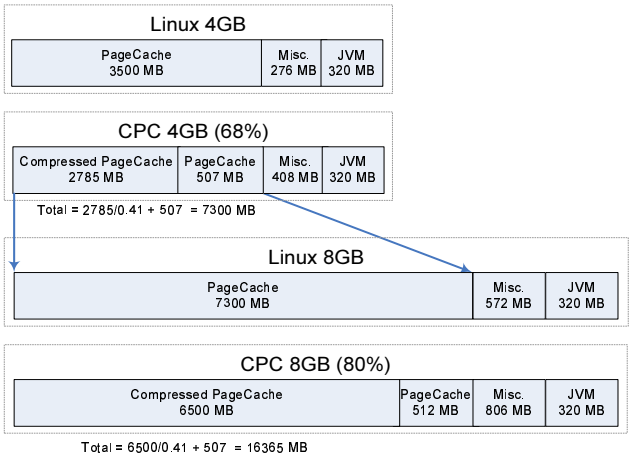

Figure 3: Compressed Page Cache Memory Layout

Name	RAM	CPC memory	PageCache Size
Linux 4GB	4GB	0%	3400MB
Linux 8GB	8GB	0%	7300MB
CPC 4GB	4GB	68%	7300MB[1]
CPC 8GB	8GB	80%	16300MB[1]

Table 1: Summary of configurations evaluated. [1]with a data compression factor of 41%

Figure 2 shows the CPC diagram. In this scenario, we capture a page that is close to being discarded from the page cache, compress it, split its content on top of the SLAB allocated buffers, and update its reference in the radix-tree to point to the new compressed page. The original page frame is discarded and the new page is inserted at the head of the LRU list. If it is not referenced in a period of time, then it is discarded by the kernel reclamation code. When a lookup on the page cache returns a compressed page, we allocate a new page frame and fill it up with the decompressed data and the SLAB buffers that contain the old data are returned to the SLAB allocator. If the allocation of the new frame fails, the compressed page frame is discarded and a null value is returned, so the kernel takes the appropiate actions to read the required data from the filesystem or swap device.

4. Experimental Results

4.1. Experimental Environment

We evaluated the performance of our compressed page cache with the Tomcat [1] web server and the SPECWeb 2005 [4] benchmark. The SPECWeb 2005 benchmark is divided into three logical components that run on different servers interconnected by a gigabit ethernet switch. The first component is the distributed client emulator that runs on a group of OpenPower 720 servers. The second is the web server that runs on a JS21 blade with two dual core Power970, 8GB of RAM, two 60GB SCSI drives and two ethernet gigabit links connected to the main switch. Finally, the third component is the database emulator (BESIM) that runs on an OpenPower 710. The JS21 server runs a 2.6.21 Linux kernel augmented with the compressed page cache (CPC), while all the other servers run a Linux distribution with a standard 2.6.9 kernel version. In this paper we focus our attention on the SPECweb2005 Support workload, which is designed to simulate a vendor's support web site. The two principal characteristics of the Support application are the use of only plain connections and a large working set per client, so the benchmark tends to be I/O disk intensive.

4.2. SPECWeb2005 Benchmark

In this section we evaluate the performance of the Tomcat web server running on top of the CPC vs running the web server on top of a plain Linux kernel by using the SPECWeb 2005 Support application. The original working set of the SPECWeb Support is generated with random data so it is incompressible. In order to evaluate the benefits of our compression approach we have replaced the content of the original working set with the content of some files from the Silesia corpus [18], which is intended to represent the current content of common diskfiles. This dataset has an average compression factor of 41% with the LZO compression algorithm that is between the ranges that some papers forecast for in memory data [19], [13] and [2]. We have chosen the LZO algorithm because it has a good compression ratio with file backed data and is one of the fastest available compressors.

We compare the performance of four diferents configurations. Firstly, we run the web server benchmark with the standard 2.6 Linux kernel configured with 4GB and 8GB of physical RAM. The results of both configurations are used as a bottom and upper baseline result to compare with the performance obtained using the compressed page cache (CPC) on a system with 4GB of physical RAM, but configured with a page cache size equal to the 8GB configuration due to the compression effect. Finally we also run the benchmark with the CPC and 8GB of physical RAM to prove it's scalability. The experiments are summarized in table 1. As we can observe the two plain Linux configurations with 4GB and 8GB of physical RAM have a page cache of size 3600MB and 7300MB respectively. The remaining RAM is used as a heap by the Java virtual machine that runs the Tomcat server and for Linux internal purpose, like network buffers and other non swappable slab caches. In figure 3 we can see a diagram detailing the memory lay-

out of the four configurations evaluated. The region labeled "misc" includes network buffers, a minimum pool of free pages, the array of struct pages and other non swappable memory. The region labeled JVM is the memory utilized by the Java Virtual Machine that runs the Tomcat web server. In order to insolate the effects of the CPC on the performance of the page cache we have configured the system to be without a swap partition, thus the anonymous memory can not be swappend out and reclaimed. We have used 68% of the memory to store compressed data on the CPC 4GB configuration in order to have a page cache as large as the plain Linux 8GB configuration. With a data compression factor of 41%, the CPC 4GB configuration with 68% of memory dedicated to store compressed data results in 2785MB of compressed data plus 507MB of uncompressed memory that adds up to a page cache size of 7300MB like the plain Linux 8GB configuration. Although the page cache size of the evaluated configurations are large, they are unable to cache all of the working set of the SPECweb Support Workload which is considerably larger. In this benchmark the working set size is proportional to the number of clients and goes from 17GB for 1000 concurrent clients to 37.4GB for 2200 concurrent clients.

4.3. Performance Results

In the first set of experiments we run a total of thirteen tests on each configuration, varying the intensity of the load from 1000 to 2200 concurrent clients in increments of 100. For each test we capture a set of performance parameters returned by the benchmark client; like the obtained throughput and the response time, as well as, a number of system metrics returned by the vmstat tool and the CPC code, e.g. disk bandwidth usage, CPU utilization, page cache size and compression/decompression times. In the second set of experiments we choose the load with the best throughput for the 8GB CPC configurations, and then vary the percentage of memory dedicated to store compressed data from 10% to 80% in increments of 10 points.

4.3.1 Compressed Page Cache vs Plain Linux Kernel

In figure 4 we see the throughput of all the configurations evaluated. As we can observe, all the configurations have a similar behavior with two different phases. In the first phase they increase their throughput linearly with the number of concurrent clients (or load). At a certain point the server stops increasing its throughput and enters the second phase which is characterized by a softly decreasing throughput as the load grows. The main difference between the configurations is the point when they get saturated i.e. the change between the first and the second phase. Figure 5 is complementary to figure 4 and shows how the response time quickly grows when a configuration reaches its saturation point. This is normal behavior observed in web servers performance [7].

As we can see in figure 4 the configuration where the standard Linux kernel has 4GB of RAM is the first to reach the saturation point with a load of 1400 concurrent clients. This low throughput can be explained by looking at figure 7 where we can check that the disk bandwidth utilization at this point is at its maximum (44MB/s). We have verified that the bottleneck is the disk bandwidth by checking the network and CPU resource usage. The standard Linux kernel configured with 8GB of RAM has a larger page cache size than the 4GB configuration as shown in figure 6. The main effect of a larger page cache is the reduction of disk accesses, saving disk bandwidth per client, delaying the saturation point and achieving better overall performance. Figure 4 shows how the 8GB configuration is able to reach its maximum throughput with 1700 concurrent clients.

The CPC configuration with 4GB of physical RAM performs somewhere between the other two setups. It is capable of obtaining a noticeably better throughput than the 4GB configuration, but is unable to reach the levels of performance of the 8GB configuration despite the fact that both configurations have the same page cache size of 7300MB as we see in figure 6. This result is explained by the disk bandwidth data plotted in figure 7 that shows how the CPC 4GB configuration is unable to exceed 41MB/s while the two configurations without memory compression reaches 44MB/s. This performance penalty is explained by how the linux memory reclamation code works.

In Linux the memory can be reclaimed through the kswapd daemon or directly by an application. In the first case, the kswapd deamon is woken up periodically and if the free memory is below a predefined threshold it starts the reclamation procedure. In the second case, the application performs a new memory allocation and if the memory is below a predefined threshold it starts the reclamation procedure. In the plain Linux kernel executing the reclamation code has no effect on performance, but with the CPC, the discarded pages have to be compressed, so the reclamation procedure is much slower. This fact slows the process of obtaining new pages to perform the required disk operations and is the cause of the lower disk read performance.

4.3.2 Compresed Page Cache Scalability

In figure 9 we can see the detailed CPU usage of the CPC 8GB configuration with a constant load of 2100 concurrent clients. The PageCache key shows the size of the CPC (sum of both compressed and uncompressed page frames), the User key reflects the CPU time spent in the Java virtual machine, the System key shows the CPU spent on system calls and the Compression and Decompression keys the amount

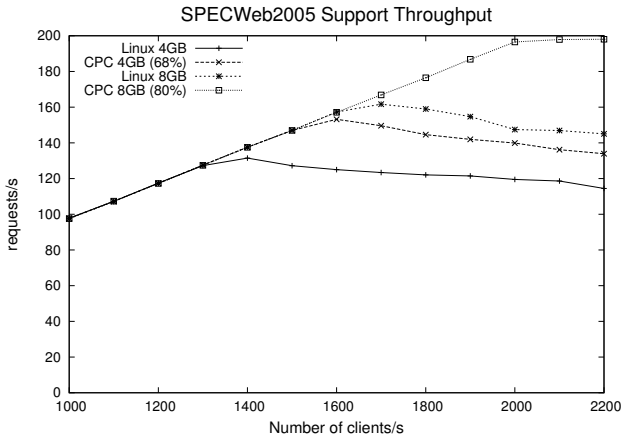

Figure 4: Throughput comparison

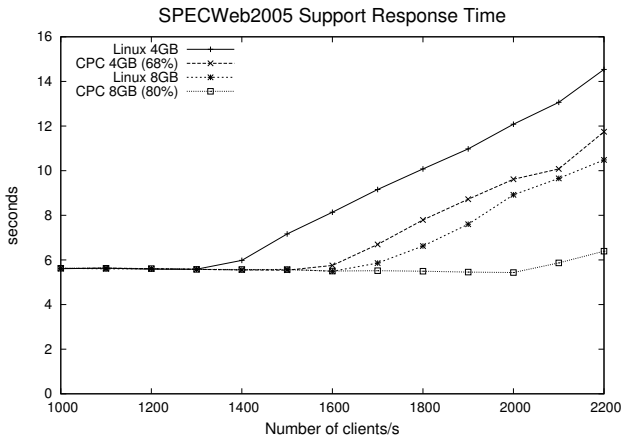

Figure 5: Response Time comparison

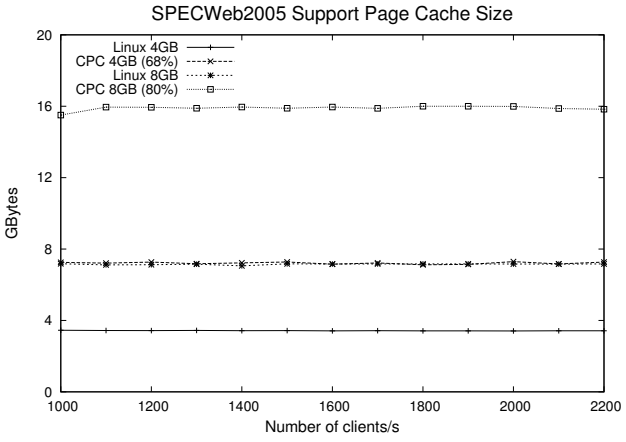

Figure 6: Linux Page Cache Size

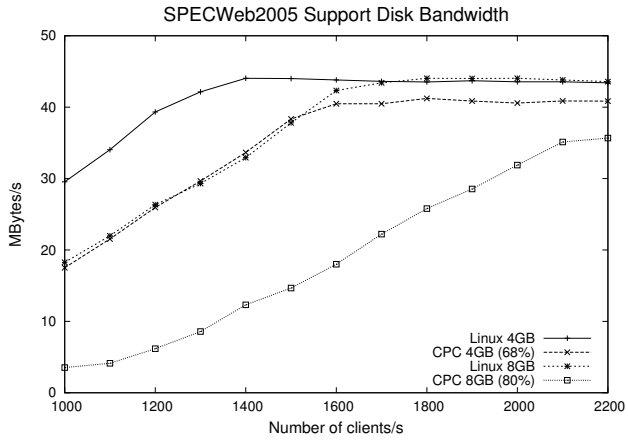

Figure 7: Disk Bandwidth Utilization

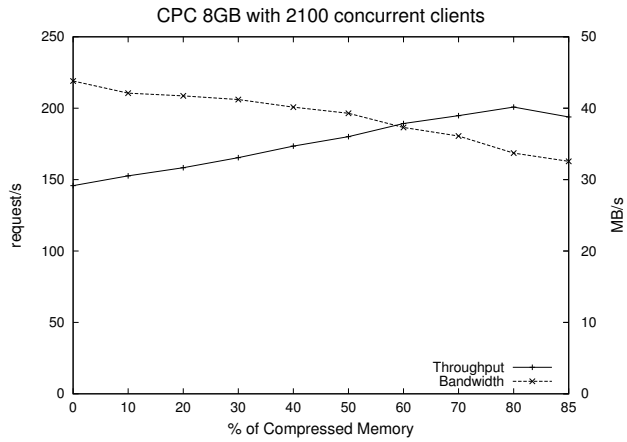

Figure 8: Throughput and Disk Bandwidth Trend

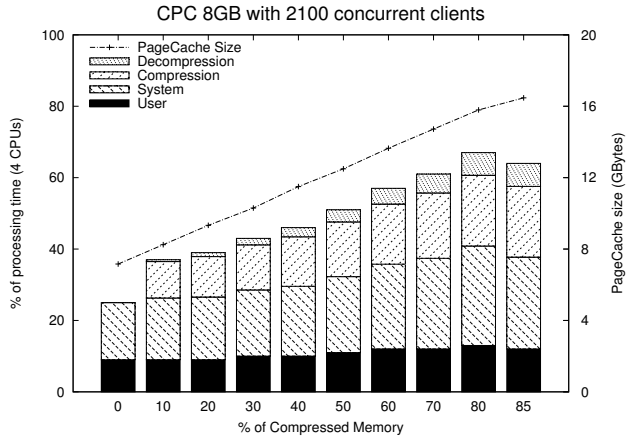

Figure 9: Detailed CPU usage

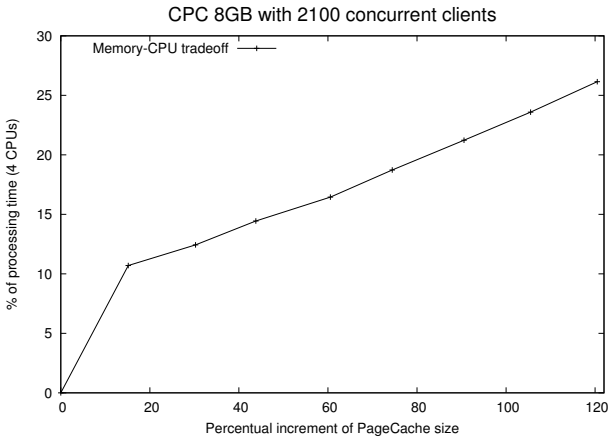

Figure 10: Memory and CPU trade-off

of CPU used for compressing and decompressing data respectively.

Figures 8 and 9 show how the CPU spent in User and System grows proportionaly to the throughput obtained. In contrast, the CPU time dedicated to compressing data has a large impact as soon as we dedicate some memory to store compressed data. From that point on, the CPU spent in Compression grows proportially to the number of decompressions, which is proportional to the number of cache hits, and also grows as the compressed page cache size grows. We have two key factors that explain the big differences between the compression and decompression times. Firstly, the compression time of a page frame is double its decompression time and secondly, we compress all the page frames when they are reclaimed, so that all the data that is read from the disk or from the compressed page cache is compressed sooner or later. In contrast, we only decompress a page when a page cache lookup has a hit; that is , for example, 40% of the time with 70% of memory dedicated to store compressed data.

We can observe how the throughput increases as the percentage of compressed memory increases and thus the PageCache size also increases. In this case, we reach the maximum throughput when 80% of the memory is dedicated to store compressed data. Figure 3 shows how this configuration has an uncompressed page cache of 512MB. Below this minimum size, the configurations start to degrade their throughput due to the memory shortage. In figure 8 we can observe how the throughput increases and the disk bandwidth decreases considerably as the percentage of compressed memory rises up to 85%, when the throughput starts to decrease. These figures show the good scalabilty that our CPC has in function of the percentage of the compressed memory used.

Figure 10 shows the trade-off between the increase in PageCache size and the computational power required to achieve it with the Support workload. As we can see the processing time required and the increment of the PageCache size are proportional. We can also observe that the sum of CPU time for Compression and Decompression tasks range from 11% to 26% (with a page PageCache size increment of 18% to 120% respectively). The ability of the CPC to double the PageCache size from less than 8GBytes to slightly more than 16GBytes shown in figures 8 and 10, produces a remarkable increase in throughput for the CPC 8GB configurations, which is depicted in figure 4.

4.4. Synchronous vs Asynchronous CPC

The implementation of the CPC used to evaluate the performance of a web server in the last section has been designed to run the compression and decompression tasks synchronously in the current thread. This design can take advantage of a multiprocessor system because the Linux kernel and the evaluated web server running the SPECWeb2005 workload are highly threaded. In contrast, if an application were single threaded, with this design, it is more difficult to fully exploit the power of larger multiprocessors systems. To solve this problem and allow the execution of compression / decompression tasks in specialized compression hardware or dedicated cores of a heterogeneous multiprocessor system like the CBE [20], we have implemented in the linux kernel a mechanism to execute these tasks asynchronously. Our framework runs on top of the workqueues facility provided by the standard linux kernel. These workqueues have a dedicated thread for each cpu that processes the tasks enqueued. Our implementation allows the kswapd daemon to send a batch of compression tasks to multiple workqueues increasing the processing parallelism. The decompression tasks are also enqueued to dedicated workqueues, but in this case we can not send multiple decompression tasks because the decompressions are always triggered one to one. We have evaluated and compared the asynchronous mechanism with the CPC 8GB configuration using the same parameters and the results have not shown a noticeable degradation of the performance of the asynchronous CPC. Despite the overhead of enqueue tasks and context switches, the performance of both mechanisms are on a par because this overhead is small compared with the time required to perform compression and decompression tasks. With these results, we can predict that the performance of the CPC running on top of specialized hardware and heterogeneous multiprocessors are both feasible and promising. We think that our software approach can be augmented with the utilization of this new kind of hardware resources to create hybrid memory compression systems that have the flexibility of the software implementations and the performance of the specialized hardware solutions.

5. Conclusions and Future Work

We implemented on the Linux OS a main memory compression system that takes advantage of the full power of current multiprocessors architectures. We evaluated its performance with a higly threaded web server running the realistic SPECWeb2005 benchmark and obtained positive results such as a 30% throughput improvement and a 70% reduction in the disk bandwidth usage. Our CPC implementation allows us to maximize the utilization of multicore and multiprocessor systems by memory-bounded applications, interchanging CPU cycles with memory space in a flexible manner. With the obtained results from our asynchronous implementation we can anticipate the big impact that this technology can have in conjunction with new multiprocessor and multicore technologies like the Niagara [14] and CELL [20] processors which have the power to accelerate the compression and decompression tasks, opening up the performance improvement to a wider set of applications bounded by the memory size or disk I/O bandwidth. Our results show how our CPC implementation can utilize almost all of the physical memory to store compressed data and improve the overall performance of the system by a large margin. In the future, we will study the benefits of sending anonymous and file-backed pages to disk in a compressed form to effectively increase the bandwidth of the disk by the data compression factor, and to reduce the number of compressions that are now required.

6. Acknowledgments

This work has been supported by the Spanish Ministry of Education and Science (projects TIN2007-60625), by the IBM SoW on Adaptive Systems, as part of the BSC-IBM collaboration agreement, and the HiPEAC Network of Excellence (IST-004408).

References

[1] *Apache Software Foundation. Tomcat Project. http://tomcat.apache.org*.

[2] B. Abali, H. Franke, D.E. Poff, R.A. Saccone, C.O. Shulz, L.M. Herger, and T.B. Smith, "Memory Expansion Technology (MXT): Software Support and Performance," IBM I. Research and Development, vol. 45, no, 2, 2001.

[3] Michael J. Freedman. The Compression Cache: Virtual Memory Compression for Handheld Computers. Technical report, Parallel and Distributed Operating Systems Group, MIT Lab for Computer Science, Cambridge, 2000.

[4] *Standard Performance Evaluation Corporation. SPECweb2005. http://www.spec.org/web2005/*.

[5] V. Beltran, D. Carrera, J. Guitart, J. Torres, and E. Ayguadé. A hybrid web server architecture for secure e-business web applications. In *HPCC*, pages 366–377, 2005.

[6] V. Beltran, J. Torres, and E. Ayguadé. Improving disk bandwidth-bound applications through main memory compression. In *MEDEA '07: Proceedings of the 2007 workshop on MEmory performance*, pages 57–63, New York, NY, USA, 2007. ACM.

[7] D. Carrera, V. Beltran, J. Torres, and E. Ayguade. A hybrid web server architecture for e-commerce applications. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, pages 182–188, Washington, DC, USA, 2005. IEEE Computer Society.

[8] R. Cervera, T. Cortes, and Y. Becerra. Improving Application Performance through Swap Compression. In *Proceedings of the USENIX Technical Conference (Freenix track)*, 1999.

[9] R. S. de Castro, A. P. do Lago, and D. D. Silva. Adaptive Compressed Caching: Design and Implementation. In *SBAC-PAD '03: Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, page 10, Washington, DC, USA, 2003. IEEE Computer Society.

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

[11] J. Guitart, V. Beltran, D. Carrera, J. Torres, and E. Ayguade. Characterizing secure dynamic web applications scalability. *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 108a–108a, 04-08 April 2005.

[12] J. Guitart, D. Carrera, V. Beltran, J. Torres, and E. Ayguade. Session-based adaptive overload control for secure dynamic web applications. *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 341–349, 14-17 June 2005.

[13] S. F. Kaplan. Compressed Caching and Modern Virtual Memory Simulation, Ph.D. Thesis, University of Texas at Austin, December 1999.

[14] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*.

[15] One Laptop per Child Foundation. *One Laptop per Child Project. http://laptop.org/*.

[16] Rodrigo S. de Castro. *Compressed Caching for Linux. http://linuxcompressed.sourceforge.net/*.

[17] S. Roy, R. Kumar, and M. Prvulovic. Improving System Performance with Compressed Memory. In *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*, page 66, Washington, DC, USA, 2001. IEEE Computer Society.

[18] Sebastian Deorowicz. *Silesia Compression Corpus. http://www-zo.iinf.polsl.gliwice.pl/ sdeor/silesia.html*.

[19] I. C. Tuduce and T. R. Gross. Adaptive Main Memory Compression. In *USENIX Annual Technical Conference, General Track*, pages 237–250, 2005.

[20] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.

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