

# Accelerating Software Memory Compression on the Cell/B.E.

Vicenç Beltran

Barcelona Supercomputing Center  
vbeltran@bsc.es

Xavier Martorell   Jordi Torres   Eduard  
Ayuadé

Barcelona Supercomputing Center  
Technical University of Catalonia  
{xavim, torres, eduard}@ac.upc.edu

## Abstract

The idea of transparently compressing and decompressing the content of main memory to virtually enlarge their capacity has been previously proposed and studied in the literature. The rationale behind this idea lies in the nature of some applications whose performance are memory or disk-bounded. For this kind of application it is acceptable to use CPU cycles to compress and decompress data on the fly, thus increasing the available memory. This additional memory capacity can allow the execution of larger applications without swapping, or can significantly reduce the number of disk access for applications with a working set that largely exceeds the main memory.

Previous studies that have worked on this idea can be classified as either software or hardware based. The software approach is usually implemented at the operating system level and works on top of commodity hardware. The hardware approach is based on modified or specialized hardware not present in current systems. The main advantage of the software approach is that it can run on unmodified commodity systems, while the hardware approach need ad-hoc hardware but it usually provides better performance for a larger set of applications. Although both approaches have been proved effective for some workloads neither of them has been widely used in production systems.

In the current scenario of many-core systems and heterogeneous processors, the flexibility of a software approach and the performance of a hardware approach can be combined to boost the real applicability of main memory compression. In this paper we propose and implement a software memory compression system for the Linux kernel, that offload the CPU-intensive compression task to the specialized processor units present in the Cell/B.E. We have evaluated our hybrid proposal with the IOzone benchmark, obtaining a 5x speedup with 80% of the system memory used as a compressed cache.

**Keywords** Memory Compression, Linux kernel, Cell/B.E.

## 1. Introduction

Main memory compressed systems are usually based on the reservation of some physical memory to store compressed data, virtually increasing the amount of memory available to the system. 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 a number of factors such as the application workload compression factor, the application access pattern (multi-threaded vs single-threaded, sequential access vs random access), the computational cost of the compression algorithm and finally the ratio of compressed/uncompressed memory configured in the system.

Previous works has mainly focused on memory compression 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 this approach the time that the application will be waiting for disk I/O is used to perform the compression and decompression of the data, thus making the most of underutilized cycles to accelerate memory access. All the afore-mentioned works have been evaluated on single processor systems so its conclusions are only relevant to single processor systems. Recently, new studies have investigated the benefits that compressed memory systems can contribute to disk I/O bandwidth bound applications like a multi-threaded web server. 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 memory available devoted to buffer cache in the system, 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 vice-versa. However, this CPU power is now more easily available with the proliferation of multi-core and multi-processor systems which can be utilized for this purpose (5). Although this approach expands the range of applications that can benefit from memory compression, its computational cost is yet too high to be widely used.

The main contribution of the paper is to prove the feasibility and suitability of heterogeneous processors such as the Cell/B.E. to accelerate the computationally expensive compression and decompression tasks. To this end, we have extended our multiprocessor aware memory compression system (5) to run on a heterogeneous processor and offload the compression and decompression tasks to the specialized coprocessor units present in the Cell/B.E. To accomplish our objective we have solved three challenging issues. First, we have re-implemented the LZO algorithm to take advantage of the vectorial nature of the SPU's present on the Cell/B.E. Second, we have extended the Linux Crypto API and the Kspu framework (17) to be able to use the coprocessors present in the Cell/B.E. from inside the Linux kernel. Finally, we have evaluated the performance of our prototype with the IOzone (20) benchmark

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

using the majority of the physical memory to store compressed data. As we will show, our proposal is able to work with optimal performance when up to 80% of the memory is dedicated to store compressed data.

The rest of the paper is organized as follows: Section 2 presents the related work on memory compression and section 3 introduces the Cell/B.E. Section 4 presents our augmented compressed page cache design for the Cell/B.E. In section 5 we discuss the results obtained with the IOzone benchmark. Finally, section 6 draws the conclusions of the paper.

## 2. Related Work

Memory Compression implementations can be classified as either hardware or software approaches, each with their own advantages and drawbacks. From the point of view of performance, the hardware approach is the best, mainly because of the utilization of custom hardware specially designed for this purpose. In contrast software approaches make the most of underutilized CPU resources present in current systems to provide a more modest performance improvement. In this paper we are focused on a software approach that uses hardware accelerators to improve the overall system performance. In the next paragraph we perform a brief review of the most relevant studies of both software and hardware memory compression techniques.

From the hardware based studies we can cite (23), (4), (9) which are all based on simulation techniques. The results of these works shows noticeable improvements in the amount of available main memory and system performance but they all need specialized hardware not available in current systems. To the best of our knowledge, the only main memory compression hardware approach implemented is the IBM MTX technology (2). In (18) the MTX technology and the operating system modifications needed to run on top of the compression hardware are described. In (3) a performance evaluation with promising results over different workloads is presented. Ultimately this project has not been a success, probably because of the cost of the specialized hardware versus the cost of memory chips.

The most relevant studies based on software techniques are now described. The first memory compression proposal, due to Wilson (21), intends to improve system performance reducing the latency associated with disk accesses. In (8) 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. (12) studied 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 (6) Cervera et al. implemented a compressed swapping mechanism to reduce the number of times the Linux OS has to access the swap device. Although the amount of compressed swap memory used was rather small, they observed a noticeable improvement of system performance. This is the first work that swaps out pages to the swap device in a compressed form, virtually increasing its capacity. Freedman et al. (10) 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 access in both power efficiency and speed by 1–2 orders of magnitude. In (15) 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 (7) De Castro et al. reevaluates the use of adaptive compressed caching to improve the system performance. The main idea behind their proposal 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 they are 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 et al. (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 (14) Nitin Gupta has ported De Castro'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 (13). The work is based on ideas from Kaplan, De Castro and Irina. Their 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 multi-core and heterogeneous systems as they are standard commodity hardware. To the best of our knowledge, our implementation is the first to fully take advantage of a heterogeneous multi-core processor to offload the compression and decompression tasks. Another remarkable characteristic is that it is highly scalable in the amount of RAM that can be used to store compressed data (up to 80% of the physical RAM).

## 3. Cell/B.E.

The Cell Broadband Engine Architecture (CBEA) (11) is a single chip heterogeneous multiprocessor. The design goals of the Cell processor were to address the fundamental challenges facing modern microprocessor development: high memory latencies and on-core power dissipation. Until now, microprocessors have achieved performance improvements through higher clock frequencies and deeper pipelines, but the fundamental problem that current processors face is the memory wall (22). On modern processors significant amounts of time are spent waiting in memory stall, due to the large difference between the processor and the memory speed. Large memory latencies make it difficult to obtain further performance gains with traditional processor designs based on hardware caches. The Cell processor approaches this problem in a different way, providing a heterogeneous processor with explicit memory management. This approach potentially improves the throughput of the processor, but it also increase the effort to efficiently implement a program.

Figure 1 shows the three basic components of the Cell processor. First, the PowerPC Processor Element (PPE), which is primarily intended to manage global resources. Second, the Synergistic Processing Elements (SPEs) that are specialized vectorial processors. Finally, the communication between the PPE, the SPEs, main memory, and external devices is realized through the Element Interconnect Bus (EIB).

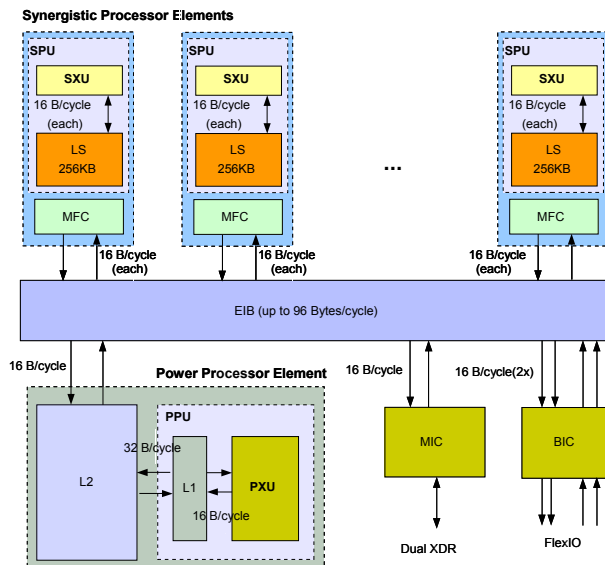


Figure 1: The Cell Broadband Engine Architecture

### The Power Processor Element

The PPE is the main Cell processor designed to be power efficient and manage all other system peripherals and processor cores. It has a dual-issue in-order execution design. It provides two hardware threads that execute simultaneously. The PPE is a 64 bits processor with a vector unit (VMX) that has the usual cache hierarchy with a 32KB first-level (L1) instruction and data cache and a 512KB second level (L2) cache used to hide memory latencies. The PPE is in charge of executing the Operating System, as well as distributing the load between all the SPE units. The communication between the PPE and the SPEs is done through shared memory regions or through direct mail boxes provided by the SPE units.

### The Synergistic Processing Element

SPEs are specialized vector units with an instruction set similar (but not compatible) to the PPE VMX. The main difference is found in the memory hierarchy, which is divided into three levels: the main memory, the local stores and a large unified register files. The large unified register file makes it possible to hold the majority of operands directly inside the CPU core without having to spill values onto the stack. A 256Kb local store is used to store the SPE code and temporary data. SPEs can perform asynchronous DMA transfers between their local stores and main memory

SPEs are designed to execute regular computationally intensive programs rather than general purpose software. This allows the system to hide memory latencies without having to employ complex hardware mechanisms such as branch-prediction, out-of-order execution, and deep pipelining, often used in superscalar processor cores. This permits the reduction of the hardware required to obtain a high throughput and hardware utilization on regular programs, but at the cost of requiring a considerable effort in order to optimize non-regular programs in such a way that an acceptable performance is obtained. The simple design of the SPE cores makes it possible to pack together up to eight SPU in a single multi-core die.

### The Element Interconnect Bus

Communication between SPEs and the Element Interconnect Bus (EIB) is realized through the SPEs Memory Flow Controller (MFC). The MFC of each SPE can enqueue up to 16 DMA commands, which implies that the whole system can process more than 100 DMAs simultaneously. It also provides memory mapped I/O registers (MMIO) and channels to monitor DMA commands, SPU events and facilitate interprocess communication via mailboxes and signal-notification. Mailboxes are a set of queues that support exchanges of 32 bit messages between an SPE and other devices. Two one-entry mailbox queues are provided for sending messages from the SPE. The EIB is a 4-ring structure (2 clockwise, 2 counterwise) for data, and a tree structure for commands with an internal bandwidth of 96 Bytes per cycle. The EIB has two external interfaces, the MIC, which is the interface between main memory and EIB and the BEI, which allows data transfer between EIB and I/O devices.

## 4. A Compressed Page Cache for the Cell/B.E.

Our original Compressed Page Cache (5) (CPC from now on) implemented in the Linux OS, was the first software memory compression implementation to fully exploits shared memory multiprocessors and multi-core systems. The objective of our design was to extend the unified page cache of Linux with a high performance CPC that fully exploits the power of current multiprocessor systems. Another design objective was to minimize the number of changes made to the Linux kernel, avoiding the addition of complex algorithms or data structures. Figure 2 shows the Linux unified page cache extended with our compressed page cache. In the next sections, we describe the original implementation and the issues that have been overcome to port our shared memory multiprocessor implementation to the Cell/B.E. processor.

### 4.1 Compressed Page Cache Design

Linux memory management is developed around its core concept: the page frame. All the memory available on the system is divided into page frames of the same size (usually 4Kb). The page frame is the smallest unit of work to manage the system memory. The con-

tent 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 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. 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 to also contain compressed page frames. Each compressed page frame has an augmented struct page called struct `cpage`, which is dynamically created to manage the content of the page. 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 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. SLAB buffers that contain the compressed 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 right actions to read the required data from the filesystem or swap device.

To extend our CPC design to support an heterogeneous architecture such as the provided by the Cell processor, we have to overcome a number of issues. First, the original design used the synchronous Linux Crypto API to perform the memory compression and decompression. We need to change to an asynchronous mode of operation, because in the Cell the compression and decompression operations are not performed on the same processor that run the Linux kernel. Second, we have re-implemented the LZO compression algorithm to make the most of the vector nature of the SPU processors. Finally, we need to extend the Kspu framework (17), which allows the execution of code on the SPUs from the kernel side, to overcome the limitation of only be able to use one SPU. In the following sections we explain in greater detail the mentioned issues.

## 4.2 Linux Crypto API

The Linux crypto API is used to publish cryptographic, compression and digest algorithms in a unified way to in-kernel users such

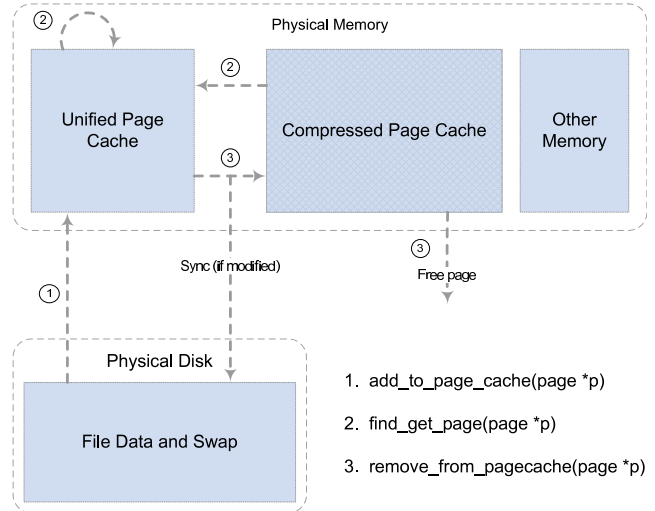


Figure 2: Compressed Page Cache Diagram

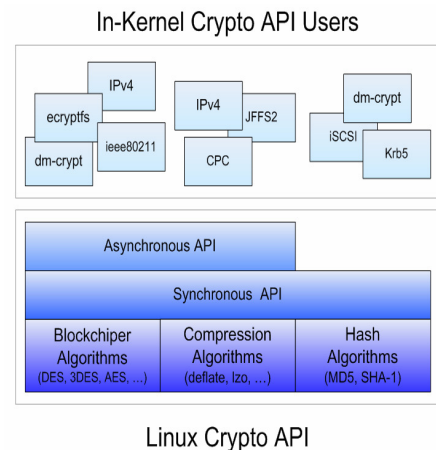


Figure 3: Linux Crypto API extended with asynchronous compression

as IPsec, some wireless drivers and file systems drivers. As we can see in Figure 3, all the encryption, compression and hashing algorithms implement the synchronous API. This API is the most frequently used inside the kernel, as its use is easy and natural. In contrast, the asynchronous API, built on top of the synchronous one, provides more advanced functionalities and was specifically designed to take advantage of specialized encryption hardware. This API provides a way to asynchronously send a batch of requests, thus improving the performance and use of specialized hardware such as cryptographic processors. The original Linux asynchronous crypto API does not support neither compression nor hashing algorithms, so our first step was to extend this asynchronous API in order to be able to use our offloaded compression algorithm on the SPUs. Then, we integrated the asynchronous compression crypto API with the Kspu framework to execute the compression and decompression on the SPU processors. This integration allows the transparent acceleration of the compression tasks to all kernel users including, but not limited to, our CPC.

### 4.3 Kspu Framework

The Kspu framework is an experimental module designed by Sebastian Siewior (17) which allows the execution of kernel code on one SPU from inside the kernel space. The primary objective is to be able to offload cryptographic algorithms from the Crypto API, but the framework can also be used to execute arbitrary code. The integration of the asynchronous compression API and the Kspu framework did not produce any major problems, as the steps needed to encrypt or compress a block of data are similar.

As an example of the Kspu framework usage, an in-kernel user enqueues a request using the asynchronous compression API. This API enqueues the request in a dedicated queue of the Kspu framework. When the SPU is ready, the request is notified to the SPU. Then the data is fetched from the SPU. When all the data required is in the SPU the compression algorithm is executed and the resulting data copied back to main memory. Finally, the Kspu framework calls a method handler of the crypto API that notifies to the client the finalization of the process.

The Kspu framework has two major limitations: the impossibility of running kernel code on more than one SPU, and the lack of request priorities, so we cannot give a higher priority to decompression requests, which are always in the kernels's execution critical-path. Both limitations have been addressed in our implementation to improve the overall system throughput and reduce the latency of decompression tasks. We link together each of the four logical PPE processors present on the QS20 (two physical hyper-threaded processors) with two different SPUs. One SPE executes the compression tasks and another SPE executes the decompression tasks triggered from their associated PPU. With this configuration the wait time of individual decompression requests is minimized, while the throughput of batch compressions requests is maximized.

### 4.4 LZO vectorization

The performance of the original LZO implementation on one SPU was three times slower than a Pentium 4 processor at the same frequency. The poor performance of the original LZO code can be explained by the specialized characteristics of the SPU cores. These cores do not have branch prediction hardware and miss-predicted branches incur high penalties. Moreover, most branches of the initial LZO code cannot be accurately predicted at compilation time. Finally, the original code was completely scalar and full of unaligned memory accesses, so it cannot capitalize on the vector nature of the SPU cores.

To improve the performance of the code we had to completely rewrite the compression algorithm to avoid, as far as possible, the most frequent branch instructions and the unaligned memory accesses, as well as, to vectorize the most time consuming functions of the compression algorithm. Some control dependencies were changed to data dependencies. In addition, the most time consuming functions, the string hash calculation and the string comparison, were vectorized to speed up the algorithm. With this optimizations we greatly improved the performance of the original LZO code as we can see in Figure 4, but we needed to completely rewrite the original code from scratch. This figure shows the performance of the LZO code with a widely used working set used to evaluate the performance of compression algorithms, which is composed of the first 100MB of the Wikipedia. Figure 4 shows the performance of the original LZO code on three different CPUs at the same clock frequency (3200 MHz): a Pentium 4, the PPU core of the Cell processor and the SPU core of the Cell processor. Finally, the figure also shows the performance of the vectorized LZO code on the SPU of the Cell processor. As we can see the vectorized code is three times faster than the original LZO code on the SPU processor and 20% faster than the original code on a Pentium 4 processor. It is worth noting that the power consumed by one SPU is a fraction

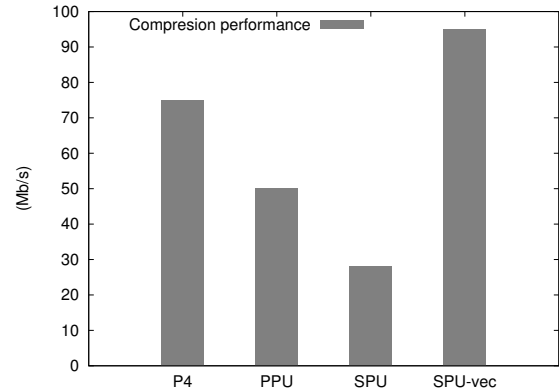


Figure 4: LZO performance on several processors

of the power consumed by an out-of-order superscalar processor such as the Pentium 4. Thus, if we take into account that the Cell processor has eight SPUs, the aggregated compression bandwidth is nearly ten times larger than the obtained with a Pentium 4 processor, which has a similar overall power consumption. The superior power efficiency of accelerators to run computationally intensive tasks makes up for the extra effort needed to port and optimize an algorithm to this heterogeneous architecture.

## 5. Experimental Results

All the experiments presented in this section have been conducted with the Linux kernel 2.6.23 augmented with our CPC running on a IBM QS20 Blade, which have two Cell/B.E. processors at 3200MHz and 1GBytes of RAM. Our original goal, as in (5), was to evaluate the performance of the SPECWeb 2005 (1) benchmark, but the network driver stops working every time we put the Blade under high network load, so we were unable to run this benchmark. We have finally used the IOzone (20) benchmark to evaluate the performance of our accelerated CPC. IOzone is a filesystem benchmark tool that generates and measures a variety of file operations. In these experiments we will focus on the performance of the read, stride-read, reverse-read and re-read file operations. The read operation measures the performance of reading an existing file not already present in the system cache. After the read operation completes, the stride-read operation measures the performance of reading the same file in blocks of 4KB and a stride of 200KB, so portions of the file could already be present on the system cache. Afterwards, the reverse-read operation measures the performance of reading the file backwards. Finally, the re-read operation measures the performance of reading a file that have been recently read.

We have configured the IOzone benchmark to run with four concurrent threads to simulate a multi-threaded application. Each thread run the four afore-mentioned file operations on a file of 512 Mbytes, so the total working set is 2 Gbytes (two times the available system memory). Each file is composed of a mix of the Silesia corpus (16), which has a compression ratio of a 40% with the LZO algorithm.

Figure 5 shows the performance of our CPC configured with 0%, 20%, 40%, 60% and 80% of the total system memory to store compressed data. The first configuration shows the performance of the plain Linux kernel, while the rest of configurations show the performance of our CPC. The left Y axis measure the bandwidth

obtained for each of the file operations while the right Y axis measures the effective (compressed + uncompressed) page cache size. For each of the five configurations the performance of the four file operations is depicted.

The performance of the read operation for the Linux Kernel is near 15 Mbytes/s. This low value can be explained because there are four threads reading four different files from the disk concurrently, so it is not possible to achieve the sequential peak performance of the disk, which is near 50 Mbytes/s. The performance of the stride-read operation is the worst, because the seek latencies introduced by the stride read greatly reduce the effective bandwidth from the disk. The reverse-read operation has half the performance of the read operation, in the plain Linux kernel, as it can not benefit from the read-ahead Linux heuristics. Note that the performance of the reverse-read operation increases with the CPC size, because the probability of finding data in the cache is higher when using a larger fraction of system memory for compressed data. Finally the re-read operation has almost the same performance of the original read operation, because the size of the four files (2Gbytes) do not fit in the page cache (less than 1Gbyte for this configuration), and the LRU algorithm replaces the content of the files that are in memory, before these files can be reused. The set of configurations with 20%, 40% and 60% of the memory used to store compressed data have a similar behavior. The read operation has the same performance that the first configuration, as each benchmark starts with the page cache empty. The re-read operation also has the same performance that the plain Linux Kernel because the sizes of the page cache (1204, 1518 and 1832 Mbytes respectively) are still smaller than the size of the working set (2Gbytes). The stride-read operation and the reverse-read operation improve their performance proportionally to the cache size, due to the increasing probability to request a portion of a file that is already in the page cache. Finally, the configuration with 80% of the memory used to store compressed data shows the best performance for all but the read file operation, which has the same performance that the rest of configurations. In this configuration more than 800 Mbytes are used to store compressed data, so with a data compression factor of 40%, around 2000 Mbytes can be cached in a compressed form. The compressed and uncompressed page cache accounts a total of 2100 Mbytes that is enough to cache all the working set. This explains the big performance increase in the reverse-read, stride-read and re-read operations that are five times higher than the obtained with the plain Linux kernel.

The results obtained show how the CPC proportionally improves the performance for the experiment in which the overall working set does not fit in memory, while it boosts the performance by a factor of five when the working set fits in memory. Moreover, for all the evaluated configurations (with and without memory compression), the overall main CPU utilization (user+systems) was between 10% and 15%, thus the CPC does not introduce a significant overhead to the main PPE processors of the Cell/B.E.

## 6. Conclusions

We implemented on the Linux OS a main memory compression system that takes advantage of the Cell/B.E. coprocessors to perform the computationally intensive compression and decompression tasks. We have optimized the original LZO compression algorithm to make the most of the vectorized processor units present in the Cell/B.E. We have also extended the Linux crypto API and the Kspu framework to be able to use several SPUs at the same time from the Linux kernel. Moreover, our evaluation has proved the feasibility and suitability of using coprocessors to offload the expensive compression and decompression tasks. The IOzone benchmark shows a speedup of up to 5x with the Linux Kernel augmented with our compressed cache. Future work will include further research on

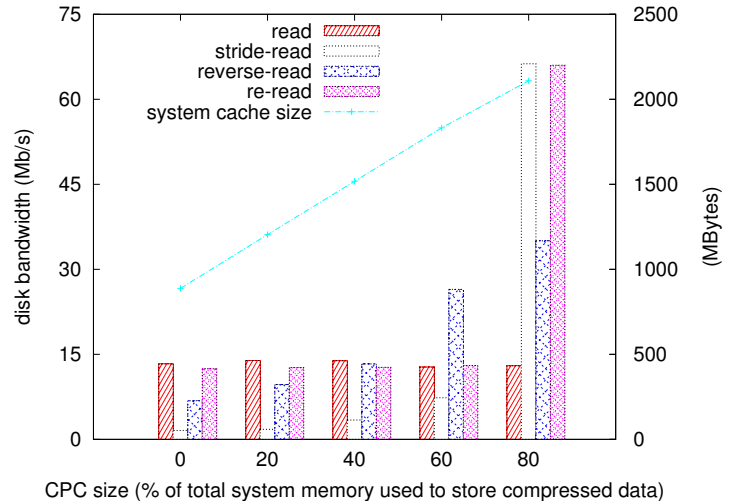


Figure 5: IOzone performance. File size 2Gbytes

the use of coprocessors to offload operating system tasks, as well as, the role of this kind of coprocessors to lower the energy costs of general purpose systems.

## Acknowledgments

Thanks to Michael Perrone and Daniele P. Scarpazza for their support and help with the LZO implementation for the Cell/B.E. processor. 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] Standard Performance Evaluation Corporation. *SPECweb2005*. <http://www.spec.org/web2005/>.
- [2] Bulent Abali, Mohammad Banikazemi, Xiawei Shen, Hubertus Franke, Dan E. Poff, and T. Basil Smith. Hardware compressed main memory: Operating system support and performance evaluation. *IEEE Trans. Comput.*, 50(11):1219–1233, 2001.
- [3] Bulent Abali, Mohammad Banikazemi, Xiawei Shen, Hubertus Franke, Dan E. Poff, and T. Basil Smith. Hardware compressed main memory: Operating system support and performance evaluation. *IEEE Trans. Comput.*, 50(11):1219–1233, 2001.
- [4] A. Alameldeen and D. Wood. Adaptive Cache Compression for High-Performance Processors. In *31st Annual International Symposium on Computer Architecture*, June 2004.
- [5] Vicenç Beltran, Jordi Torres, and Eduard Ayguadé. Improving web server performance through main memory compression. In *ICPADS '08: Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 303–310, Washington, DC, USA, 2008. IEEE Computer Society.
- [6] R. Cervera, T. Cortes, and Y. Becerra. Improving Application Performance through Swap Compression. In *Proceedings of the USENIX Technical Conference (Freenix track)*, 1999.
- [7] Rodrigo S. de Castro, Alair Pereira do Lago, and Dilma Da 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.
- [8] Fred Dougliis. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *USENIX Winter*, pages 519–529, 1993.

- [9] Magnus Ekman and Per Stenstrom. A Robust Main-Memory Compression Scheme. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 74–85, Washington, DC, USA, 2005. IEEE Computer Society.
- [10] Michael J. Freedman and Recitation Rivest Tr. The compression cache: Virtual memory compression for handheld computers. Technical report, Technical report, Parallel and Distributed Operating Systems Group, MIT Lab for Computer Science, Cambridge, 2000.
- [11] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.
- [12] Scott Frederick Kaplan. *Compressed caching and modern virtual memory simulation*. PhD thesis, The University of Texas at Austin, 1999. Supervisor-Wilson,, Paul R. and Supervisor-Fussell,, Donald S.
- [13] One Laptop per Child Foundation. *One Laptop per Child Project*. <http://laptop.org/>.
- [14] Rodrigo S. de Castro. *Compressed Caching for Linux*. <http://linuxcompressed.sourceforge.net/>.
- [15] Sumit Roy, Raj Kumar, and Milos 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.
- [16] Sebastian Deorowicz. *Silesia Compression Corpus*. <http://www-zo.iinf.polsl.gliwice.pl/sdeor/silesia.html>.
- [17] Sebastian Siewior. *Diploma thesis: Acceleration of encrypted communication using co-processors*. <http://diploma-thesis.siewior.net/html/>.
- [18] R. Brett Tremaine, T. Basil Smith, Mike Wazlowski, David Har, Kwok-Ken Mak, and Sujith Arramreddy. Pinnacle: IBM MXT in a Memory Controller Chip. *IEEE Micro*, 21(2):56–68, 2001.
- [19] Irina Chihaiia Tuduca and Thomas R. Gross. Adaptive Main Memory Compression. In *USENIX Annual Technical Conference, General Track*, pages 237–250, 2005.
- [20] William D. Norcott and Don Capps. *Iozone Filesystem Benchmark*. <http://www.iozone.org>.
- [21] Pe Re 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.
- [22] Wm Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. Technical report, University of Virginia, Charlottesville, VA, USA, 1994.
- [23] Keun Soo Yim, Jihong Kim, and Kern Koh. Performance Analysis of On-Chip Cache and Main Memory Compression Systems for High-End Parallel Computers. In *PDPTA*, pages 469–475, 2004.