Improving System Performance with Compressed Memory

Sumit Roy, Raj Kumar, Milos Prvulovic[1],
Client and Media Systems Laboratory
HP Laboratories Palo Alto
HPL-2001-111
April 24th , 2001*

E-mail: sumit@hpl.hp.com

system architecture, compression, performance evaluation Linux, memory optimization

This report summarizes our research on implementing a compressed memory in computer systems. The basic premise is that the throughput for applications whose working set size does not fit in main memory degrades significantly due to an increase in the number of page faults. Hence we propose compressing memory pages that need to be paged out and storing them in memory. This hides the large latencies associated with a disk access, since the page has to be merely uncompressed when a page fault occurs. Our implementation is in the form of a device driver for Linux. We first evaluate a number of compression algorithms for use in the driver for Linux. We then show results with some applications from the SPEC 2000 CPU benchmark suite and a computing kernel. It is seen that speed-ups ranging from 5% to 250% can be obtained, depending on the application behavior.

* Internal Accession Date Only Approved for External Publication?
Presented at the IEEE 15th International Parallel and Distributed Processing Symposium, 23-27 April 2001, San Francisco, California.
[1] Currently at University of Illinois at Urbana-Champaign, IL, work performed while at HP Labs

ABSTRACT

This report summarizes our research on implementing a compressed memory in computer systems. The basic premise is that the throughput for applications whose working set size does not fit in main memory degrades significantly due to an increase in the number of page faults. Hence we propose compressing memory pages that need to be paged out and storing them in memory. This hides the large latencies associated with a disk access, since the page has to be merely uncompressed when a page fault occurs. Our implementation is in the form of a device driver for Linux. We first evaluate a number of compression algorithms for use in the driver. We then show results with some applications from the SPEC 2000 CPU benchmark suite and a computing kernel. It is seen that speed-ups ranging from 5 % to 250 % can be obtained, depending on the application behavior.

1 Introduction

This paper summarizes our research on implementing a compressed memory in computer systems. Due to ever increasing size of application programs, modern operating systems use the concept of virtual memory to extend the visible size of the real random access memory (RAM) by backing it with a region on a permanent storage device (swap file or device) [1]. This process also allows multitasking systems to run multiple time-sliced processes on the machine. An active process can be given access to the complete virtual address space of the processor. Idle processes can be swapped out to disk, and kept ready to run when their turn arrives again. The address space is divided into page frames. A translation mechanism is used to convert the virtual address issued by a running process to the physical page that contains the instructions or data required by the process. When the system runs low on physical pages, pages that have not been used in the recent past can be written to the swap device. The newly available page frames can now be supplied to active processes.

When the page frame on the swap device is required at a later time by a process, a page fault occurs and the data has to be fetched explicitly from the swap device. The problem is that the throughput for applications whose working set size does not fit in main memory degrades significantly due to an increase in the number of page faults. Disk access latencies are of the order of tens of milliseconds, which is much larger than memory access time which are of the order of tens or hundreds of nanoseconds. Hence many researchers have proposed compressing memory pages in preference to swapping them out to disk [2, 3, 4, 5, 6]. This hides the large latencies associated with a disk access, since the page has to be merely uncompressed when a page fault occurs. The compressed memory system can be implemented in multiple ways, including software approaches, such as modifications of the operating system kernel, as well as hardware implementations, for example using compressed cache

lines. The former approach requires access to kernel source code and may not be easily ported across different operating systems. A hardware implementation adds to the cost of a machine. If the approach uses an enhanced memory controller, it may only be suitable for new machines.

Our system consists of a loadable device driver (currently for Linux). This approach is readily portable to other operating systems like WindowsNT, Windows 2000 and HP-UX. Another advantage of our system is that the driver can simply be unloaded for those applications that do not benefit from memory compression. This can be done on a running system, and does not require a shutdown. We have evaluated the performance of our system with some of the larger applications from the SPEC 2000 CPU benchmark suite and a computational kernel. It is seen that for these applications, speed-ups from 5 % to 250 % can be obtained when compared to execution without memory compression.

The rest of this report is organized as follows. Section 2 explains the architecture of the memory compression scheme. Section 3 describes the actual implementation of the driver. The selection of the compression algorithm is discussed in Section 4. Our experimental results are shown in Section 5. Section 6 reviews related research in this field. We conclude with some ideas for future work in Section 7.

2 Compressed Memory Architecture

In order to understand the operation of the compressed memory we briefly review the standard paging operation of a system. As memory pressure builds up, the system paging daemon starts to move pages out to the swap device to maintain a pool of free pages in memory, as shown in Figure 1.

At a later time, if the page that was written to disk is accessed, it has to be explicitly fetched from the swap device and put into a free physical page frame. This results in a page-fault and causes a page-in operation as shown in Figure 2.

This disk access is multiple orders of magnitude slower than direct memory access ie. tens of milliseconds for disk accesses and tens or hundreds of nanoseconds for access to memory. The operating system can try to hide the latency of this disk access by switching to another runnable process, or by aggressively prefetching pages from disk. In many environments, such as engineering workstations dedicated to one main task (EDA/CAD/CAM), there may not be another process that can be run to hide disk access latencies. On the other hand, prefetching of pages from disk will work perfectly only if one can exactly predict the page reference pattern of the application. This is not possible or prohibitively expensive in terms of computational resources for the general case. Swap daemons do try to capitalize on the locality of reference that

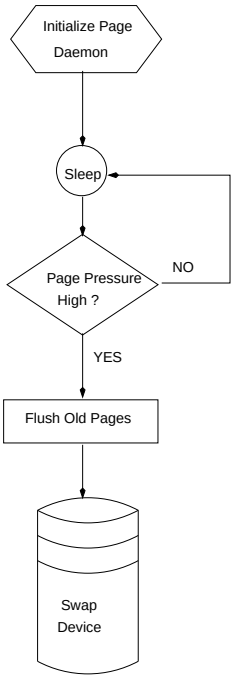

Figure 1: Paging out under Memory Pressure

most application programs exhibit by clustering page-ins. However, our experimental results show that there is still considerable room for performance improvement via memory compression.

The idea of memory compression is to hide the disk latencies by storing swapped out page frames in a compressed form, but still in physical memory. On a subsequent page fault, the page can be quickly decompressed and supplied to the application program. The physical memory is divided into two parts, uncompressed memory and compressed memory. The uncompressed memory caches frequently used pages from the compressed memory. The compressed memory in turn caches pages from the swap device. This part of the memory is managed by a kernel device driver. Detailed operation of the system is described in the next section.

3 Implementation Details

The actual implementation of the compressed memory system is in the form of a loadable device driver module. This module provides support for compressed in-memory caching of block devices. The main benefit from it comes when it is used on swap devices, but it can be used on filesystem devices as well. In the Linux implementation the module replaces the 'linear' personality of the Multiple Device Driver. The Multiple Device Driver is a block device driver that sits on top of other block de-

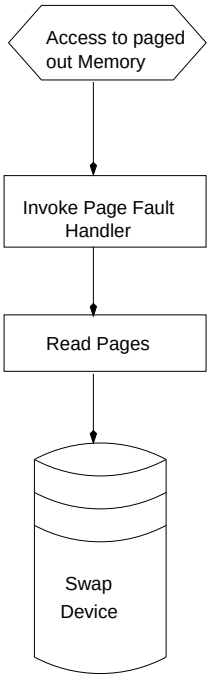

Figure 2: Page Fault on access to Page Frame on Swap Device

vices and provides additional functionality, such as disk concatenation and various levels of RAID. To add a new functionality, modules called "personalities" are used; examples of personalities are 'raid0', 'raid1' and 'raid5'. Personality 'linear' allows several block devices to be logically concatenated and presented to the system as one, larger, device. We could have added a new personality for our caching compressed memory; however, the Multiple Device Driver, which is a Linux kernel module, has a hard-coded list of all personalities and adding a new one requires changes in the code. Instead, we completely replace one of the personalities with ours, which results in a fully stand-alone package that does not require changes to the rest of the kernel code.

The system architecture consists of the device driver, the driver memory for caching compressed pages, a swap device, and the regular uncompressed memory, as shown in Figure 3.The device driver appears as a regular block device to the system once it is loaded. This block device is formatted as a swap partition and added as a swapping device to the system. When the operating system tries to swap out a page, it will initiate a write on this special swap device. The device driver intercepts the page and compresses it. It then copies the compressed page to the previously reserved memory area. When the process tries to access the page sometime in the future, the device driver intercepts the read request to the swap device and decompresses the page. Decompression of a page is much faster than reading data from a disk. This translates into reduced total execution time for the application.

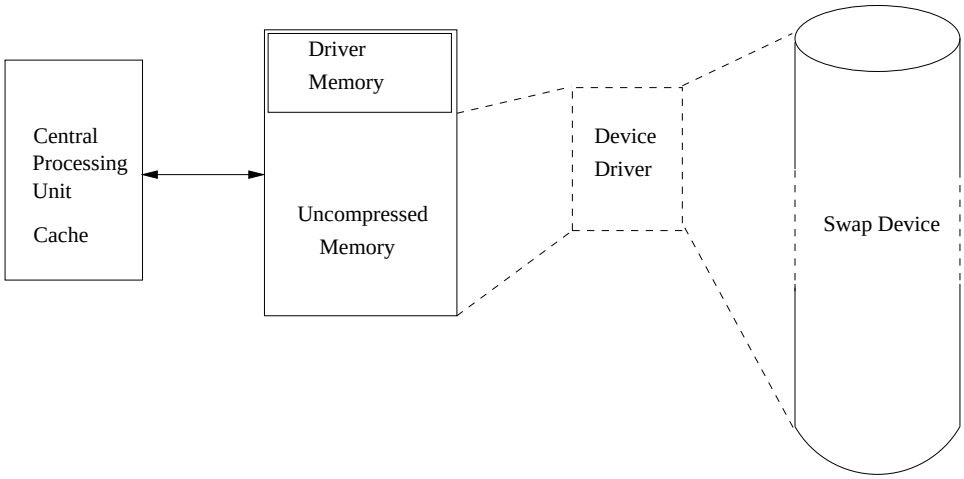

Figure 3: Architecture of Memory Compression System

The algorithm used is as shown in Figure 4. The device driver initially allocates some kernel memory, based on load time parameters. This memory is carved into fixed size buckets (256 Bytes in the current implementation). The buckets are chained together into a linked list, this forms the initial free list. The size of swap file that this device represents is also specified at load time. This value is used to create a page table, with one entry per swap page. The page table entry includes a pointer to the start of the bucket chain in which the compressed page is held and the size of the compressed page.

The amount of memory required by the compressed page depends on the contents of the page as well as the memory compression algorithm. In some cases the compressed data can actually be larger due to some constant data overheads from the compression algorithm. For such cases we use a dummy compression algorithm, which only makes a copy of the page. The memory compression algorithm used for a page is encoded in the page table entry. If the compression achieved for a large number of pages is very poor, it is possible that there is not enough memory reserved to store them. At that time, the driver should not allocate more memory, since the system is already under severe memory pressure, and it could lead to instability. Hence the driver starts to write compressed pages to disk, using a history of accesses to determine which are the less frequently used pages. A special entry in the page table indicates whether the page is on the disk. Pages on disk are stored in an uncompressed form. If a later access to a page misses in the compression cache, the page can be supplied from the disk without having to decompress it.

The software solution makes the implementation very flexible. Additional and improved compression algorithms can easily be added. It is also possible to tune the algorithms that allocate memory buckets and write back pages to disk based on appli-

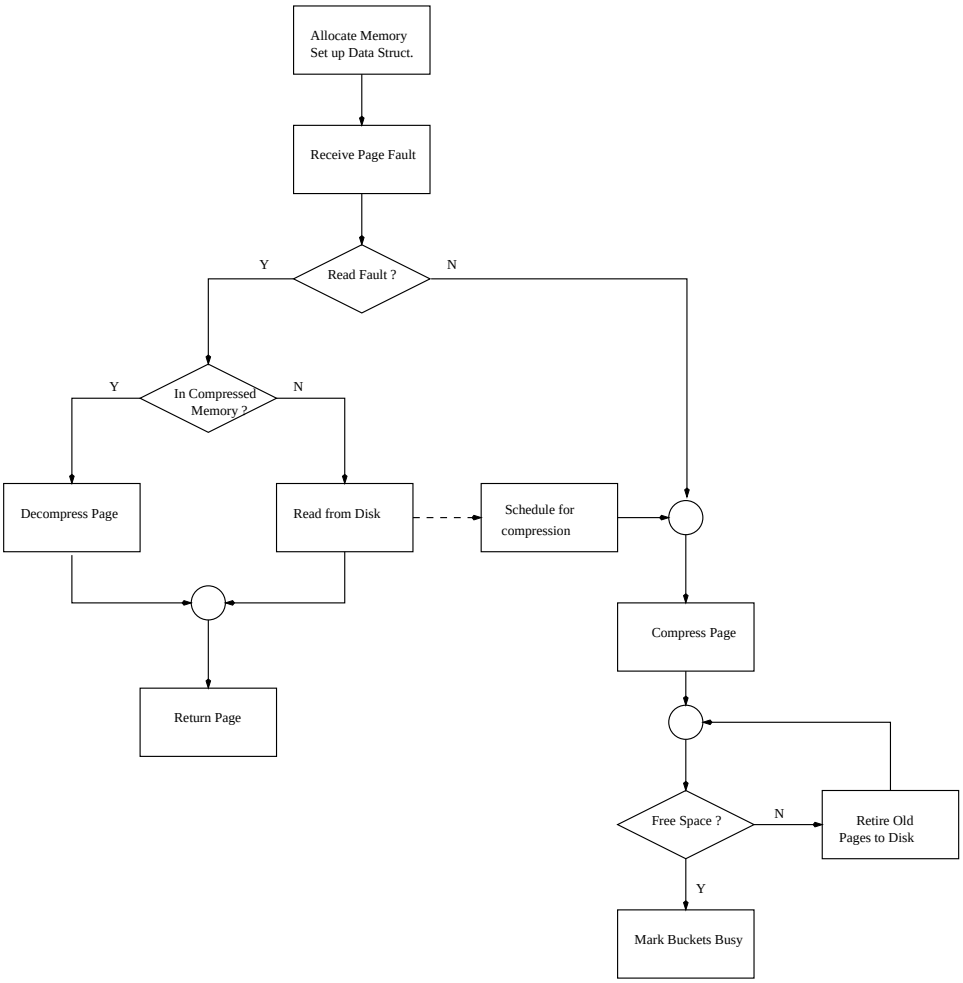

Figure 4: Flowchart of Memory Compression Algorithm

cation behavior. Unlike prior attempts, our system does not require modifications to the operating system kernel, nor does it require hardware support from the computer. This of course does not preclude the use of hardware compression accelerators for further improving the performance. An additional advantage of our system is that the driver can be dynamically loaded and unloaded. Memory compression schemes can lead to poor performance in case of applications that compress substantially worse than the assumed 2x ratio. Since the overall memory accessible to the application is reduced by the factor reserved for storing compressed pages, it results in more page faults. If the faults have to access the disk due to poor compression ratios, the application runs slower than if there were no memory reserved for compression. It may even be possible that the very act of reserving memory for the compressed pages forces the working set of pages out of the main memory into the disk. In such a case, one would see a considerable slowdown in the application. However in our approach, the driver can simply be unloaded, so that all the memory is available to

the operating system for management. Thus the compression cache can be loaded and unloaded without requiring system shutdown.

4 Compression Algorithm Selection

The first step in evaluating the effect of memory compression was to select a suitable compression algorithm. These tests were carried out on an HP Kayak XA with a 500 MHz Pentium III processor and 128 MBytes of memory. The operating system is Linux 2.2.16, based on the S.u.S.E 7.0 distribution. The memory compression algorithms include *zlib*, *lzo*, *lzrw*, *WKdm*, and *WK4x4*. Details about the algorithms are available in [4]. All the compression algorithms were compiled without any compiler optimization. The intent was to capture the the intrinsic quality of the compression algorithm, without considering coding or compiler characteristics.

The input data sets were obtained by taking regular snap-shots of the swap file while running a large visualization application. The interval between captures was 10 minutes. The application used was a computer vision program that does 3D scene-reconstruction from multiple 2D images [7]. The data in the snap-shot was then compressed with the various algorithms using 4096 byte chunks as input. This corresponds to the page size of the system. Each compression speed data point is obtained by averaging the time to compress that chunk over 10 runs. The compression ratio, as well as the decompression speed for 4096 byte pages, are similarly obtained from multiple runs.

Figures 5 - 7, show the histograms for the compression ratio, the compression speed and the decompression speed. The histograms are obtained by counting the data points in 100 buckets, which are evenly spaced between the logarithms of the minimum and maximum values of each set. Note that the axes are plotted on a logarithmic scale.

It is clearly seen from Figure 5 that most pages compress by a factor of about 2, irrespective of the algorithm used. Only a few pages are found to lie in the extremes of having very high or very low compression ratios. It is seen that *zlib* has the best compression ratio, followed by *WK4x4* and *WKdm*.

However, the compression ratio alone is not the only criterion for selection. During a page fault, the decompression of the page lies in the critical path, and thus its speed is significant. As seen in the decompression speed curve in Figure 6, there is a much greater variance between various algorithms. Clearly, *zlib* is the slowest algorithm.

Finally, page compression typically occurs only when a page is swapped out. The compression speed curve is shown in Figure 7. In this case *zlib* is an order of magnitude

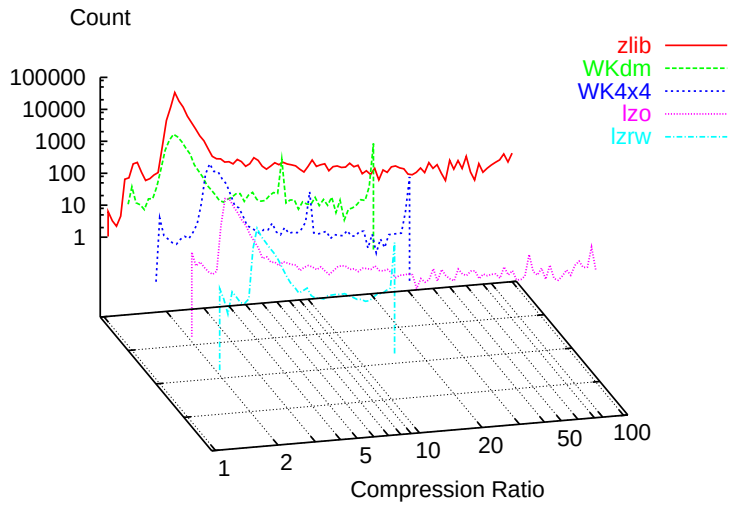

Figure 5: Histogram of Compression Ratios for different algorithms

slower than the other algorithms. This is likely due to the complexity of the algorithm, ie. the additional cost of the Huffman encoding. Based on these results, *WKdm* and *WK4x4* are the best compromise in terms of compression ratio, decompression speed, and compression speed.

5 Experimental Results

The compression scheme was evaluated on an HP Kayak XU800 with dual 733 MHz Pentium III processors (256K full speed L2 cache) and 256 MBytes of RDRAM. The swap device was a 9.1GB SCSI Ultra3 10krpm Hard Disk Drive. The operating system was Linux 2.2.16 from the SuSE 7.0 distribution. The memory compression algorithm used is WK4x4 which is described in detail in [4].

The applications tested consist of several programs from the SPEC 2000 CPU benchmark suite. The resident sizes obtained by running `top(1)` on a system with 256 MBytes are shown in Table 1. Note that some programs have a fairly large fluctuation in resident size during their execution and thus the sizes are shown as ranges.

The values for the integer benchmarks were also reported in [8] and agree well with those shown here. Since our objective was to run applications with moderate to large memory footprints we restricted further investigations to `wupwise, swim, applu`, and `apsi` from the floating point benchmarks, and `gzip, gap, vortex`, and `bzip2` from the integer benchmarks.

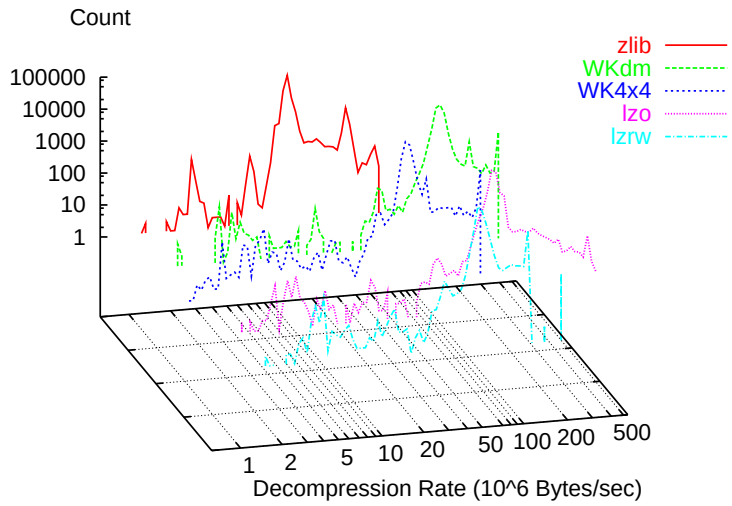

Figure 6: Decompression Speed for different algorithms

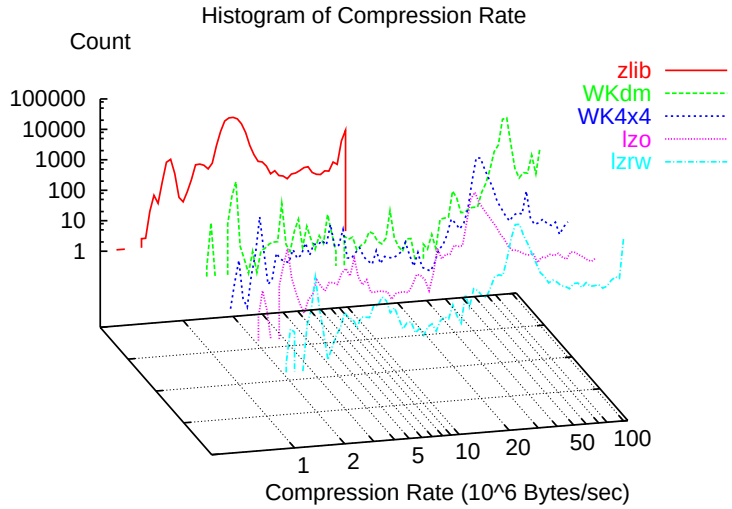

Figure 7: Compression Speed for different algorithms

Integer		Floating Point	
Program	MBytes	Program	MBytes
gzip	180	wupwise	176
vpr	2 – 35	swim	190
gcc	7 – 31	grid	55
mcf	11 – 79	applu	180
crafty	2	mesa	8
parser	26	art	3
eon	1	equake	41
perlbmk	1 – 107	amp	13
gap	192	apsi	191
vortex	48 – 122		
bzip2	179 – 184		
twowolf	2		

Table 1: Resident Size of SPEC 2000 Benchmark Programs (256 MBytes System)

Most UNIX like systems including Linux will try to make use of all available memory by dividing it between the kernel data structures, application requirements and space for buffering files. This means that, given sufficient memory, even pages that are used rarely by an application will continue to reside in memory till the program terminates. Hence we also measured the dynamic working set sizes for these programs by executing them on the same machine, but with varying memory sizes, from 64 MBytes to 256 MBytes. Figure 8 shows that the actual dynamic working set size for applications like `bzip2, gzip`, and `vortex` is less than 64 MBytes and that these programs are not likely to benefit from memory compression in systems with more than 128 MBytes of memory. Due to the strict reporting requirements of the SPEC consortium, we cannot show the actual execution times. Instead, we have normalized the results to the execution time on a machine with 256 MBytes of memory. Since `swim` had a large amount of paging in the 128 MByte configuration and did not complete successfully even after a very long time, we did not use this program any further.

Program	Total (MBytes)	Compressed (MBytes)
wupwise	96	16
apsi	160	48
gzip	64	8
gap	160	32
bzip2	64	16

Table 2: Memory Configuration for SPEC 2000 Evaluation

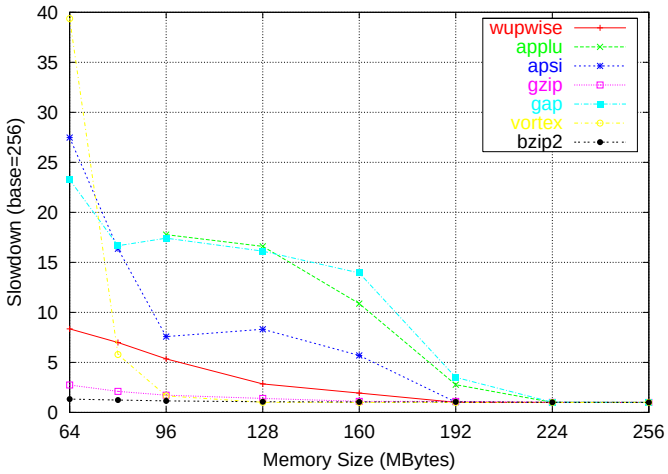

Figure 8: Effect of varying Memory Size on several SPEC 2000 Benchmarks

The programs that we tested for performance improvements had different working set sizes. The machine configuration for the speedup tests for each program is as shown in Table 2. The amount of compressed memory required to give us the best results is also shown in the table. Figure 9 shows the overall speedup that was obtained with each program. The execution time in this case is normalized to the execution time on a machine with the same memory size, but no memory reserved for compression. It can be seen that for these programs we get speedups ranging from 5 % for `bzip2` to 75 % for `apsi` when compared to the uncompressed case. The optimal fraction of physical memory that should be reserved for compression lies between 10 % and 30 % for our set of experiments.

Since the SPEC 2000 benchmarks have a fixed memory footprint, we also tested our system with a scalable computational kernel, Successive Over Relaxation. This program performs a nearest neighbor averaging algorithm on a rectangular matrix and is typically used while solving partial differential equations. To keep the elapsed time tractable, the machine size was reduced to 64 MBytes. The results are shown in Figure 10. The execution times are again normalized to the time when run without memory compression. The results confirm that the optimal fraction of memory that should be compressed lies around 25 %. The constant overhead of about 14 MBytes (ie. about 20 % of the memory in this machine) imposed by kernel data structures cause some non-intuitive behavior. For example, once 75 % of the memory is reserved for compression, even a program that requires as little as 20 % of the memory incurs a very large number of page faults. Consequently this leads to an overall slowdown in the application. Similarly, speedups are seen when the required size of the program equals the total physical memory of the machine and 6.25 % to 50 % of the memory is compressed. The kernel overhead causes the program to page severely when it requires 64 MBytes in the noncompressed case. When memory compression is introduced, the

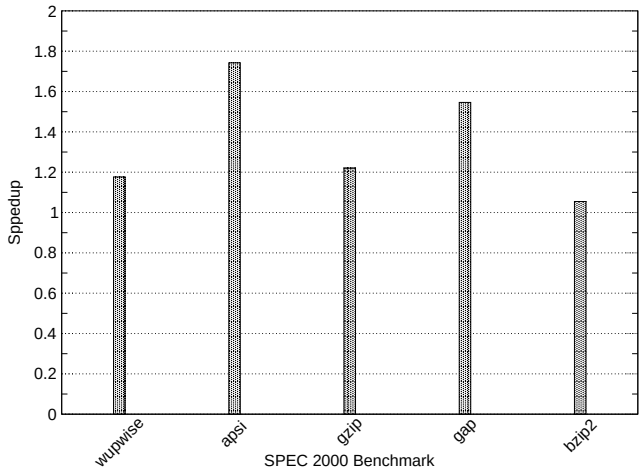

Figure 9: Speedup of SPEC 2000 Benchmarks with Memory Compression

paged out frames fit in the compressed memory, and the majority of accesses to the swap disk are saved.

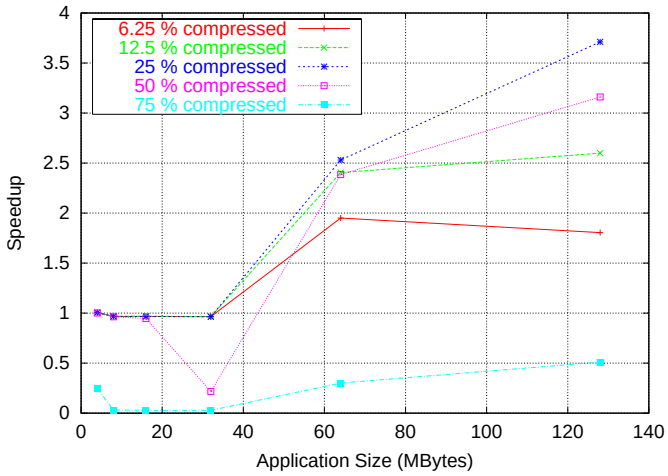

Figure 10: Scaling the Successive Over Relaxation Benchmark (64 MBytes machine)

6 Related Research

An early study of using compression in main memory targeted mobile computers with limited memory and network bandwidth, as well as small or no local disks [2]. The compression scheme was implemented by extending the dynamic trading of memory between the virtual memory system and the disk buffer cache in the Sprite Operating

System. The compressed memory region was introduced as a third consumer of memory. The results were poor due to low compression ratio (using the LZRW1 [9] compression algorithm) and restricted I/O, which only allowed transfers in 4k blocks. Though the system allowed dynamic adjustment of the size of compression memory, some of the application with compression ratios less than 2:1 still showed a slow-down. On the other hand, applications which compressed about 3:1 showed improvements by factors $1.6 - 2$. The author proposed that a greater disparity between processor speed and bandwidth would be required to show speedups. Our implementation does not require modifications to the operating system. We also use a fixed fraction of reserved memory which reduces the complexity of the implementation.

A model of main memory with data compression was developed in [3]. Standard memory hierarchy modeling techniques were extended to include the effect of compressed memory. The model was then evaluated with memory traces, assuming software, hardware and no compression. The results showed that software compression alone could give a system improvement of a factor of two, while hardware assisted compression improved performance by up to an order of magnitude. However, the model optimistically assumed a constant compression ratio of 2:1. The authors further advocate that hardware memory compression alone can provide the compression bandwidth for across the board improvements in the system. They quote a figure of 100 MBytes/s for an FPGA based implementation of a hardware compressor [10]. Our own results show that with modern high performance processors, similar compression and decompression speeds can be obtained purely in software.

Novel compression algorithms suited for compressing in-memory data representations were introduced in [4]. These modified Ziv-Lempel algorithms were shown to be very fast and symmetric in terms of the time taken for compression and decompression. They use a small 64 byte dictionary and operate on 32 bit words (unlike typical LZ compressors that use dictionaries of many kBytes and operate on 8 bit bytes). They also tracked the recent compression behavior of programs to dynamically adapt the amount of compressed memory to different phases of the program. Unlike the LRU system in [2] they recorded how well the replacement policy worked by looking at evicted pages. The scheme was evaluated using a trace driven simulator and showed that compressed caching can eliminate 20 % to 80 % of the paging cost. We make use of the WK4x4 compression algorithm developed by these authors and show performance improvements on an actual system.

A hardware compressed random access memory (C-RAM) was analyzed in [11]. The authors considered the implications on the directory structure and storage allocation design required for a hardware implementation. The approach is further investigated in [6] where an additional level 3 cache is interspersed between a completely compressed main memory and the level 2 cache. The compression engine is managed by

the memory controller in this design. As outlined in [12], operating system support is required to maintain system stability in the face of poor compression ratios. Some of the solutions include zeroing out pages that are no longer used, adjusting the paging threshold based on feedback from the compression engine, and kicking off high priority kernel threads to steal cycles from the application program which gives the system time to stabilize. Since this solution requires an additional level of caching and a new memory controller, it us only suitable for new computers. Our results on the other hand show that one can get performance improvements using a pure software approach. The device driver design can be incorporated in future as well as existing systems.

7 Conclusion and Future Work

This report summarizes our research on implementing a compressed memory in computer systems. We propose compressing memory pages that need to be paged out and storing them in memory. This hides the large latencies associated with a disk access, since the page has to be merely uncompressed when a page fault occurs. Our implementation is in the form of a device driver for Linux. We show results with SPEC 2000 benchmarks and a computational kernel applications. It is seen that speed-ups ranging from 5 % to 250 % can be obtained when compared to execution without memory compression. It is also seen that the optimal fraction of memory that should be reserved for compression lies around 25 % across a wide range of application types.

Our current work involves evaluating different compression algorithms for obtaining better performance. We also plan to investigate the behavior of our system in the presence of multiple large applications running at the same time. Finally, we plan to implement dynamic sizing of the compressed memory region to adapt automatically to different application behaviors.

Acknowledgments

We would like to thank the following HP lab members for their help in this work: Ken Wilson and Sujoy Basu for their many insights and the fruitful discussions. Bruce Culbertson and Craig Wittenbrink for making their memory intensive programs available for study. Gadiel Seroussi and Marcelo Weinberger for their expert input on compression algorithms. Tom Malzbender and Fred Kitson for their managerial support of this project.

Bibliography

[1] B. Goodheart and J. Cox, *The Magic Garden Explained*, ch. 3. Prentice Hall, 1994.

[2] F. Douglis, "The Compression Cache: Using On-line Compression to Extend Physical Memory," in *Winter 1993 USENIX Conference*, pp. 519–529, USENIX Assoc, 1993.

[3] M. Kjelsø, M. Gooch, and S. Jones, "Modelling the Performance Impact of Main Memory Data Compression," in *Proceedings 12th UK Computer and Telecommunications Performance Engineering Workshop* (J. Hillston and R. Pooley, eds.), pp. 169–184, Department of Computer Science University of Edinburgh, September 1996.

[4] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis, "The Case for Compressed Caching in Virtual Memory Systems," in *Proceedings of 1999 USENIX Annual Conference*, pp. 101–116, USENIX Assoc, June 1999.

[5] J.-S. Lee, W.-K. HOng, and S.-D. Kim, "Design and Evaluation of a Selective Compressed Memory System," in *International Conference on Computer Design*, pp. 184–191, IEEE, 1999.

[6] C. Benveniste, P. Franaszek, and J. Robinson, "Cache-Memory Interfaces in Compressed Memory Systems," Tech. Rep. RC 21662, IBM Research Division, T.J. Watson Research Center, February 2000.

[7] W. B. Culbertson, T. Malzbender, and G. Slabaugh, "Generalized Voxel Coloring," in *Proceedings of Vision Algorithms Theory and Practice Workshop*, September 1999.

[8] B. Abali, H. Franke, D. E. Poff, and T. B. Smith, "Performance of Hardware Compressed Main Memory," Tech. Rep. RC 21799, IBM Research Division, T.J. Watson Research Center, 2000.

[9] R. N. Williams, "An Extremely Fast Ziv-Lempel Compression Algorithm," in *Data Compression Conference*, pp. 362–371, April 1991.

[10] M. Kjelsø, M. Gooch, and S. Jones, "Performance Evaluation of Computer Architectures with Main Memory Data Compression," *Journal of Systems Architecture*, vol. 45, pp. 571 – 590, 1999.

[11] P. A. Franaszek and J. T. Robinson, "Design and Analysis of Internal Organizations for Compressed Random Access Memories," Tech. Rep. RC 21146, IBM Research Division, T.J. Watson Research Center, October 1998.

[12] B. Abali and H. Franke, "Operating System Support for Fast Hardware Compression of Main Memory Contents," in *Memory Wall Workshop, 27th Annual International Symposium on Computer Architecture*, pp. 1–14, June 2000.