A very fast algorithm for RAM compression

Luigi Rizzo
Dipartimento di Ingegneria dell'Informazione, Università di Pisa
via Diotisalvi 2 – 56126 Pisa (Italy)
email: l.rizzo@iet.unipi.it

DRAFT – March 1, 1996

Abstract

Many systems suffer from lack of main memory, a problem which is usually overcome by using a disk as a secondary storage device. Latency and bandwidth in retrieving pages from disk are the main disadvantages of this approach. Also, in the case of portable/low power devices, physical limitation on the size, weight and power consumption of the secondary storage device add to the list of disadvantages.

In this paper we discuss the use of compression techniques to make a more effective use of the available RAM, thus reducing the need for secondary storage. We show that in many cases memory pages contain highly compressible data, with a very large amount of zero-valued elements. This suggests the use of very fast compression algorithms based on static Huffman codes, rather than adaptive algorithms such as LZRW1 and similar ones.

We present one such algorithm which, paired with a careful layout of the data, is able to compress 4KB pages at 40MB/s even when implemented in software on an inexpensive Pentium 100 system. We also show that the algorithm can achieve interesting compression ratios despite its simplicity.

Using the proposed compression algorithm, it becomes possible to use a small portion of the physical memory as a fast, low latency, compressed swap device. This technique can greatly reduce the latency of page faults, thus improving program's performance.

1 Introduction

In many cases computer systems suffer from shortage of main memory (RAM). This derives from a number of reasons, including cost, size, power or architectural limitations, software bloat, or problem's size. The lack of RAM is usually circumvented by using a secondary storage device (usually a disk), and by implementing a virtual memory (VM) system. There are cases where using a disk as a secondary storage device is not feasible because of excessive size, weight or power consumption. Portable computing devices are an example where all the three motivations above apply. Even in absence of such constraints, latency and bandwidth in moving data to/from the disk pose problems. A disk has an access latency in the 1..10ms range (or greater), and its bandwidth is usually in the 1..10MB/s range, which is at least one order of magnitude lower than the memory bandwidth of a system. As a consequence, a page fault will block a process for some milliseconds at the very least. While the CPU can be kept active on another process while a page fault is being serviced, the latency adds up to the total execution time for the process experiencing the fault.

In this paper we investigate the usage of compression techniques to improve the effectiveness of use of main memory. The underlying idea is that part of the RAM can be set aside as a swap area, the RAM-swap. Pages being swapped out are first compressed and, if they achieve a significant compression rate, copied to the RAM swap instead of going to disk. The goal is to reduce the latency in accessing paged-out data. Of course there are tradeoffs in this approach, as the introduction of compression takes away some amount of processing and storage from an already resource-poor system. Swapping to disk keeps the disk busy and the CPU essentially free, while compressing to RAM keeps the CPU busy and the disk free. In absence of background activity, compression might be helpful. When there are other tasks in the system, swapping to disk might be wiser, so as to leave the CPU free to proceed with other tasks. But it might well be the case that the swap device becomes a bottleneck because too many processes are using it at once, in which case compression might become favorable again. In general, it is not easy to determine when the use of compression techniques can be beneficial, as it depends on a number of factors, such as the type of program, the relative speeds of CPU and disk, the system's load.

The use of compression has been already proposed in the past, applied to program code [6, 8, 9, 10], RAM/swap area [1, 2, 4], and especially the file system [5, 3]. Compressed file systems are also available in a number of commercial operating systems. In particular, Douglis [4] has done a full implementation of compressed swap in the Sprite OS. He uses LZRW1 [7], an adaptive compression algorithm which achieves good compression rates but is very time consuming (yet faster than other adaptive algorithms). Douglis concludes that the effectiveness of this technique is highly variable, and often it can be detrimental to performance if compressing a page is not significantly faster than retrieving it from disk. Thus the technique seems especially suitable to portable equipment, where speed is not the primary concern.

One of the main obstacles in the use of compression is the CPU overhead associated with the compression algorithm. Doing compression by means of a separate hardware device is a possible approach, and is being pursued by other researchers [5]. However, general exploitation of compression should preferably rely on the main processor only. This will be discussed in detail in this paper. We show how a very fast and reasonably effective compression algorithm can be implemented in software on general purpose processors. Its speed and compression rate are discussed, and a possible application within the VM system is suggested.

2 Compression

Compression techniques, both lossy and non lossy, are widely used in all fields of information processing. The compression ratio varies greatly, perhaps over two orders of magnitude, depending on the technique used and the type of data which is acted upon. The introduction of compression in the VM system obviously requires non-lossy algorithms. Its application is twofold. First, compressing data before transferring them to disk reduces the use of swap space. Secondly, some amount of RAM can be set aside to store compressed pages, instead of saving them to disk. If the compression/decompression algorithms are sufficiently fast, there might be a significant reduction in the service time for a page fault. In order to determine the applicability of the method, we need to do a preliminary evaluation on the compressibility of pages, find a suitable compression algorithm, and evaluate its performance on existing architectures.

2.1 VM features

Memory hierarchies comprise one or more stages of semiconductor memories, followed by mechanical secondary devices such as disks. With appropriate processor support, a virtual memory system is usually implemented so that pages of memory can be swapped in and out of secondary storage depending on the current needs. Reclaiming a page from secondary storage requires a significant amount of time, during which the process must remain idle. This can be partly amortized if the system has other activities which can be carried on, as it is often the case in multiuser systems. But personal computing devices (including personal workstations) usually have a limited number of concurrent activities. As a consequence, a reduction in the latency of page faults is important, and can be pursued with the help of the main processor.

Aside from file buffers, memory can be partitioned in three groups: program's text, statically initialized data, and dynamically allocated data. The size of each group varies with time, and with the type of system. In embedded systems, programs' text often take the majority of the space, while in multiuser operating systems it is likely that dynamic data areas gets the largest share. Text pages are readonly, often shared by many processes (either running the same program or using the same shared library). Paging out text pages has no cost as they are already present in the filesystem. Also, depending on the number of users of a text page, swapping it out might be a bad idea. Text pages can be compressed offline and saved to disk in compressed format. The compressibility of text pages is often an index of the compactness of the code of the target processor. It is generally not very high, around a factor of 2, possibly larger for RISC than CISC processors. Compression of text pages is of interest in embedded systems [6], as it can reduce the ROM size, the bus width, and the clock speed for a given performance level. Text compression is usually performed in short blocks, so that decompression can take place on the fly within the prefetch queue inside the processor.

2.2 Swap analysis

Data pages (both static and dynamic) are usually read-write, and as such they cannot be shared among processes. For the same reason, if they must be paged out, they require some dedicated space on the swap area. Data pages must be compressed online; their compressibility is highly variable, but often very high. In fact, it is common for for programs to overallocate memory, or to use it to store sparse data. Also, the granularity of the VM system often leaves large amounts of unused space at the end (or the beginning, if it is part of the stack) of a page. This suggests that compression of these pages can be highly effective, if either the size of the used data is known, or the page is filled up with a known pattern. This frequently happens in practice: pages are usually zero-filled before being allocated, both for robustness (many programs often incorrectly assume that memory is zero-filled) and for security reasons (as it is undesirable that a process can read data belonging to the previous owner of the page).

We have done some experiments to determine the compressibility of pages which have been copied to the swap space on disk – remember that this area only contains data pages. Our tests have been done on different Unix systems used as personal workstations (thus running several X applications) and departmental mail gateways (running several instances of sendmail and text-based mail readers). In all cases, the systems have been selected so that the swap usage were at least comparable to the amount of RAM. This varies from the 8MB of the author's system running FreeBSD, to the 64MB of an Alpha workstation. Compression has been done separately on 4KB blocks, which are a common value for the page size.

In Figure 1 we present the results of one of our experiments. We have compressed pages with gzip, a modified LZRW1, and the zero-removal algorithm presented in this paper. The graph shows the cumulative distribution of the size of compressed pages, rounded to 64-byte boundaries. The area above the graph is proportional to the compression ratio. Compressing swap pages with `gzip` yields an average compression ratio of 3-6 times, mainly depending on the type of system and its use. While the absolute compression ratios change from system to system, the shape of the curves does not. Using gzip, compression levels between -1 and -9 usually produce a small difference in the compression ratio, while the execution time significantly increases with the level. It is interesting to note that, while the various algorithms differ in the total compression ratio, the behavior is very similar on the left side of the graph, showing the existence of a relatively large number of highly compressible pages.

Further investigation on the frequency of symbols in the swap area has shown the presence of a large number of zero-valued entries. On a 32-bit architecture, the percentage of zeroes is almost the same if we consider 8- or 16-bit entries. The percentage for 32-bit entries is only marginally lower, and can be explained with the presence of word-padded entries. The high frequency of zeroes was expected, as pages tend to be zero filled, nevertheless this is an important hint to be used in the design of an efficient compression/decompression algorithm.

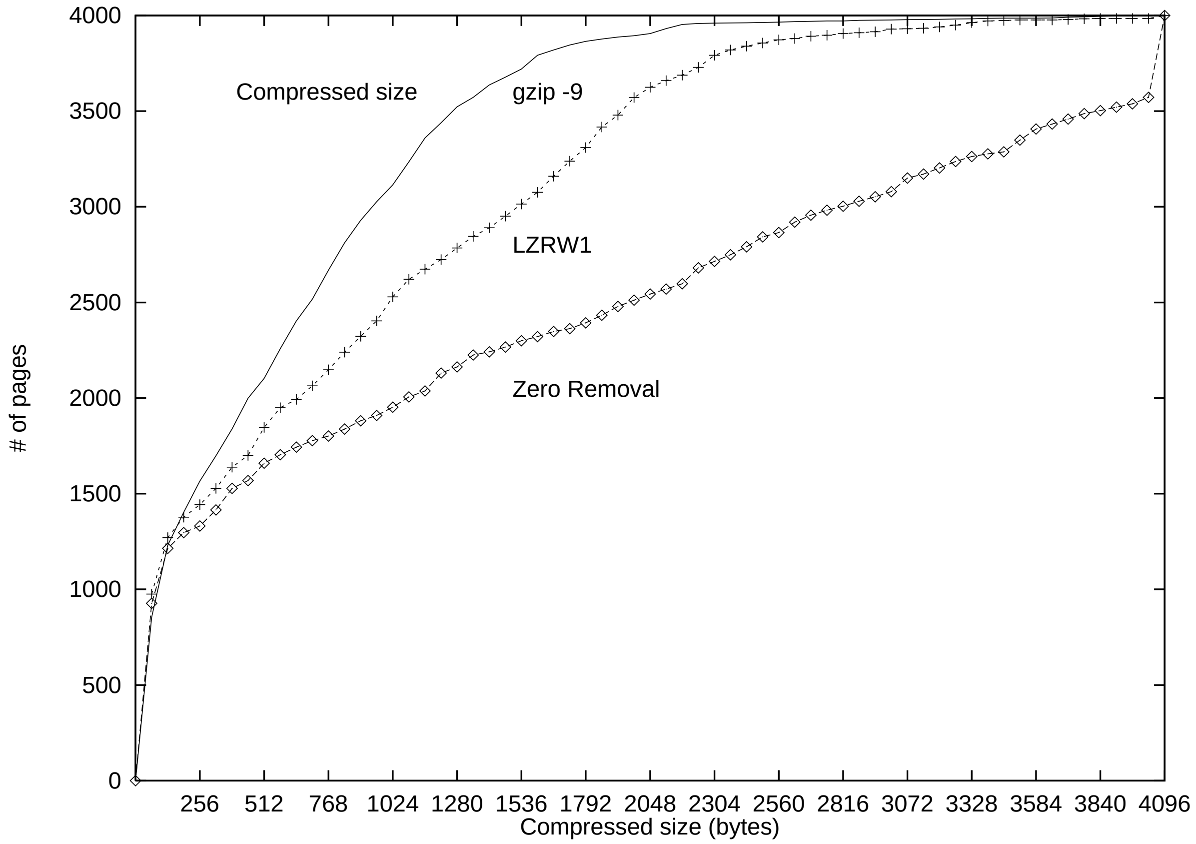

Figure 1: The cumulative distribution of compressed page sizes using different algorithms. While this graph shows the results of a single experiment, the shape of the graph does not change significantly using different systems.

3 A very fast compression algorithm

While gzip gives an interesting compression ratio, its operating speed is not acceptable: on a 486/66 it takes about 10ms to compress a 4KB page. Decompression is much faster, perhaps by a factor of 10, but the compression overhead makes the use of gzip unacceptable for our purposes. The high cost of compression is due to the fact that gzip, and similar adaptive compression algorithms [11], must search through a table of already-seen values to find a matching substring. The algorithm can be speeded up [7], but its inherent complexity remains. Gzip and other adaptive compression algorithms are based on the assumption that there are repetitive, but unknown, patterns in the data to be compressed. If the actual patterns are known, faster algorithms can be developed basing on Huffman codes. The codeword table can be built offline, and both compression and decompression become a simple substitution of codewords.

In our case, as already discussed, zero-valued symbols are very frequent. This suggests a very compact encoding for this particular symbol. Unfortunately, the next frequent symbols are much more difficult to determine, and very content-dependent. For ascii text, blanks are likely to come second. For integer numbers, small values are good candidates. For floating point data, the same might be true except that truncation and rounding might cause a lot of noise in the least significant bits of the entries. For a page containing addresses, we expect the high bits of these addresses to be very repetitive, with 2-3 different patterns corresponding to the different areas (code, stack, possibly data) of the program. These considerations make it difficult to build a general purpose, static compression table offline. Thus we only exploited the first property, namely the high frequency of zero-valued entries, to design a very fast compression algorithm. In order to improve the operating speed, the algorithm uses W-bits symbols, where W is the size of a machine-word.

We use an extremely simple Huffman encoding, where zero-valued words are represented by a single bit (0), while other words are represented by $1+W$ bits (1 followed by the word itself). This encoding is particularly interesting as it permits to deal with large symbols without the need for a separate codeword table. Also, a single test is required to produce the codeword. The only difficulty arises from the use of variable-size codewords, which are not not handled efficiently by most processors. As a consequence, codewords are packed as follows: a `bitmap` is built containing one bit per symbol, followed by a `data` area containing only non-zero symbols. This preserves the alignment of operands while still providing a reasonably compact encoding.

The above algorithm can achieve a maximum compression ratio equal to W. On most current systems, $W = 32$. From our experiments, however, it turned out that a significant number of pages could be easily compressed further than that. As a consequence, the bitmap is also compressed, using the same criterion, into a smaller `directory`. This time, the symbol size is 8 bits.

A final remark must be done on the layout of compressed page (Fig 2). We want to avoid too many accesses to memory for each symbol, both during compression and decompression. Also, sequential accesses to the memory are desirable to exploit the features of the memory architecture (page-mode

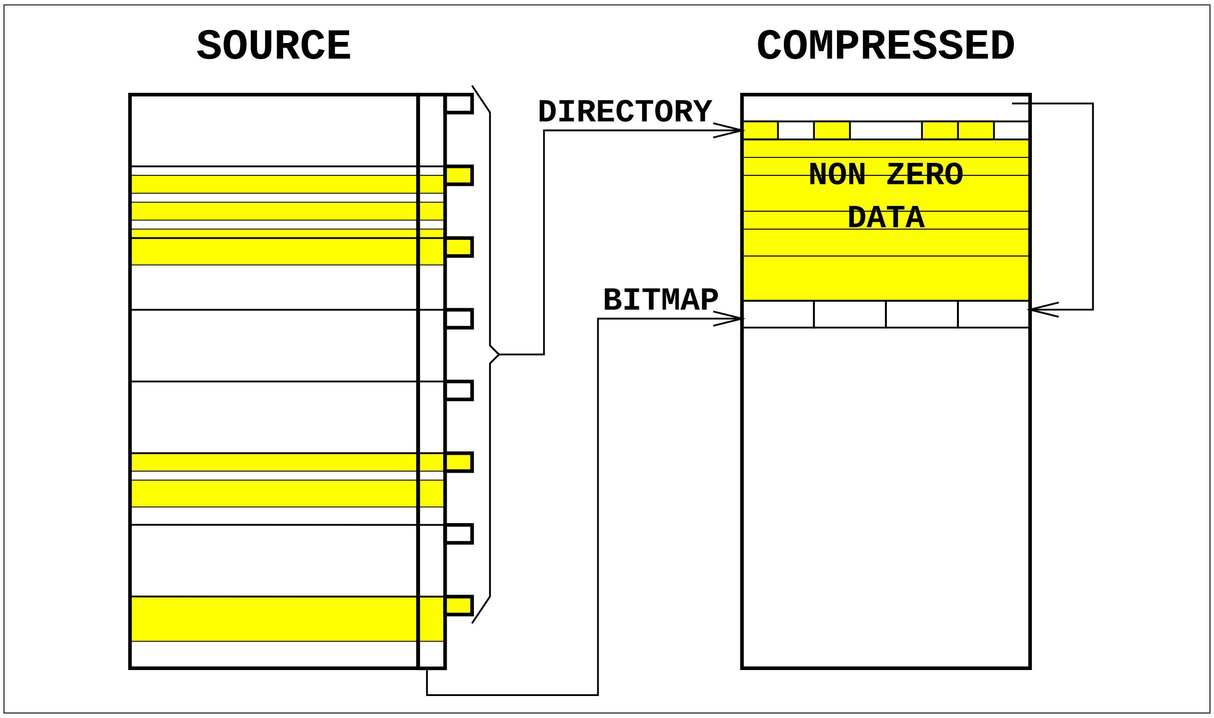

Figure 2: The layout of compressed swap pages and their relation with the source data. Grey areas represent non-zero words

CPU	OS	Compr.	T_{comp}	T_{decomp}
Alpha266	OSF	39%	78	32
Pentium-100	FreeBSD 2.1R	40%	101	118
486/100	FreeBSD 2.1R	38%	182	140
486/66	FreeBSD 1.1.5R	36%	250	180
R2000	Ultrix 4.1	60%	1500	820

Table 1: The compression ratio and operating speed of the proposed algorithm on different systems. Times are in μs per 4KB page.

DRAMs, burst accesses). The `directory` is built last, but it has a fixed size. We have placed it near the beginning of the compressed page. Both the `data` area and the compressed bitmap have variable size. The `data` area is built first, and is possibly the largest one in the compressed page. Thus we want to build it directly to its destination, rather than moving it at a later time. This forces us to place it right after the directory. Finally, we compress the bitmap and copy non-zero elements to its destination after the data area. The first word in the compressed page is used as a pointer to the beginning of the compressed bitmap. The final layout of the compressed page is shown in the right side of Figure 2.

3.1 Performance

We have implemented the compression algorithm (and the corresponding decompression one) described in the previous section, and evaluated its performance on some architectures. The experiments have been done by reading from a relatively large portion of the swap area (8MB or more) to avoid erroneous timings due to the presence of caches or other interfering phenomena. The code used in our experiments is written in C, without special optimization (apart some obvious loop unrolling), and is compiled with `gcc -O2`.

Table 1 reports the average compression ratio and, especially, the compression and decompression times, in μs per 4KB page. As it can be seen from the table, compression is reasonably fast even on obsolete systems, and extremely fast on modern CPUs, even inexpensive ones. The remarkable speed of the algorithm (over 40MB/s on a Pentium 100, much higher on the Alpha) derives from its extreme simplicity. Memory is read sequentially, and memory accesses are interleaved with some CPU instructions. This enables the use of fast page mode for the RAM and burst bus transfers, and makes a very effective use of the memory bandwidth. The differences in compression speed between the 486 (32-bit memory bus) and the Pentium (64-bit memory bus) are largely due to the different memory bandwidths. Decompression speeds are closer because of non optimized code which writes 32-bit values.

The effectiveness of the proposed algorithm compared to gzip and LZRW1 can be derived from Figure 1. The figure clearly shows that the latter two compress better than our algorithm, but their execution time is at least 10 times higher. It is noticeable that there is no significant difference in the behavior on highly compressible pages (those resulting in a compressed size between 64 and 128 bytes), which constitute a significant fraction (between 15 and 25% in our experiments) of the total number of pages.

4 Applications

The above preliminary results show that swap compression is feasible, with some advantages, even on systems with fast disks. Of course, an actual implementation of RAM-swap must address other issues, such as memory management and fragmentation. In fact, compressed pages become variable-size objects, which might complicate their management.

Given the substantial compression which can be achieved on a reasonable fraction of the pages, one possibility would be to use compression only on these pages, leaving others to the usual swap devices. This way of operating does not completely exploit the advantages of compression, but is extremely simple to implement, because compressed pages become fixed size objects, and the RAM-swap can be managed similarly to existing swap partitions. Also, attempts to compress pages can be aborted early, by gaining a further factor of 2 in the compression speed.

We have assumed that every time a page must be saved to disk, compression is attempted, and the page only goes to disk if compression is unsuccessful. Even with a fast algorithm such as the one shown in this paper, the compression costs are non negligible, and unsuccessful attempts should be avoided if possible. One way to reduce the overhead consists in keeping an additional flag for each page, indicating that compression should not be attempted on that page. The flag is cleared when a page is mapped into a process' space, and set at the first unsuccessful compression attempt for that page. This way we minimize the wasted effort, under the assumption that the entropy of a page increases with time.

The availability of compression in the VM system has implications on application programs. As an example, applications might become less careful in allocating memory (e.g. for storing sparse data structures) because the VM system will be able to take care of this. This in turn might render data pages even more compressible, and possibly simplify the program's code.

5 Conclusions and future work

We have shown that swap pages are highly compressible due to the presence of many zero-valued entries, and have presented an extremely fast compression algorithm which can compress pages at 40MB/s when implemented in software on a Pentium100. While our algorithm has a worse average compression ratio than gzip or LZRW1, it can achieve the same – extremely high – compression on a significant number of pages, and is at least 10 times faster than the other two algorithms.

The operating speed of the proposed algorithm make it suitable for the implementation of a compressed main-memory system, where part of the swap area is kept compressed in RAM. Future work will include the study of improvements to the algorithm to achieve better compression while still operating at high speed, and the performance evaluation of a compressed memory system based on the proposed algorithm.

Bibliografia

[1] A.W.Appel, K.Li, *Virtual Memory primitives for user programs*, in *Proc. of the ASPLOS IV Conference*, pp.96-107, Santa Clara, CA, April 1991

[2] D. Banks, M. Stemm, *Investigating Virtual Memory Compression on Portable Architectures*, Internal Report, Computer Science Division, U.of.C. at Berkeley, available as `http://www.cs.berkeley.edu/~stemm/classes/cs262.ps.gz`

[3] M. Clark, S. Rago, *The Desktop File System*, Proceedings 1994 Summer Usenix, pp. 113-124, 1994.

[4] F. Douglis, *The Compression Cache: Using On-line Compression to Extend Physical Memory*, Winter 1993 USENIX available as `http://www.research.att.com/orgs/ssr/people/douglis/papers/ccache.ps.gz`

[5] M. Kjelso , M. Gooch, U. Simm, S. Jones, *Hardware Data Compression & Memory Management for Flash-Memory Disks*, Proceedings Sixth International Symposium on IC Technology, Systems & Applications (IEEE), 1995.

[6] M. Kozuch, A. Wolfe, *Compression of Embedded System Programs*, ICCD '94, (Dept. of EE, Princeton University)

[7] Ross N. Williams, *An extremely fast ZIV-Lempel data compression algorithm*, in *Data Compression Conference*, pp.362-371, April 1991

[8] A. Wolfe, A. Chanin, *Executing Compressed Programs on An Embedded RISC Architecture*, SIGMICRO Newsletter, Vol. 23, No. 1-2, pp. 81-91, 1992.

[9] S. Segars, K. Clarke, L. Groudge, *Embedded Control Problems, Thumb, and the ARM7TDMI*, IEEE Micro, pp. 22-30, October 1995.

[10] M.Taunton, *Compressed Executables: an Exercise in Thinking Small*, Proceedings USENIX Summer '91, pp. 385-403, 1991.

[11] J.Ziv, A.Lempel, "A Universal algorithm for Sequential Data Compression", IEEE Trans. on Information Theory, Vol.23 pp.337-343, 1977