Algorithms and data structures for compressed-memory machines

by P. A. Franaszek
P. Heidelberger
D. E. Poff
J. T. Robinson

An overview of a set of algorithms and data structures developed for compressed-memory machines is given. These include 1) very fast compression and decompression algorithms, for relatively small fixed-size lines, that are suitable for hardware implementation; 2) methods for storing variable-size compressed lines in main memory that minimize overheads due to directory size and storage fragmentation, but that are simple enough for implementation as part of a system memory controller; 3) a number of operating system modifications required to ensure that a compressed-memory machine never runs out of memory as the compression ratio changes dynamically. This research was done to explore the feasibility of computer architectures in which data are decompressed/compressed on cache misses/writebacks. The results led to and were implemented in IBM Memory Expansion Technology (MXT), which for typical systems yields a factor of 2 expansion in effective memory size with generally minimal effect on performance.

1. Introduction

This paper gives an overview of a set of algorithms and data structures developed for systems with main-memory compression. In such systems, essentially all data are maintained in compressed form, and decompressed on access. The advantage is a potentially substantial improvement in price/performance, since the memory is typically the most expensive component in the central electronic complex (excluding disk storage) in server-class machines. The approaches described here are incorporated in IBM Memory Expansion Technology (MXT) [1], which for typical systems yields a factor of 2 expansion in the effective memory size, with generally minimal effect on performance.

It is well known that the contents of memory are generally compressible. This can occur because of repeated patterns in data or programs, or alternatively because pages may not be entirely filled; that is, they may have long strings of zeros. Such compressibility traditionally is exploited in a number of ways, including compression of files to be sent to disks or transmitted over networks, and also maintaining substantial portions of main-memory contents in compressed form. The latter approach can be seen in the IBM System/390* (for example see [2]), for which a primary use is in

©**Copyright** 2001 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

0018-8646/01/$5.00 © 2001 IBM

compressing database contents, as well as in some embedded RS/6000* processors for compressing object code [3]. For these cases, the data are known or scanned in advance, so that appropriate compression dictionaries or tables can be preconstructed.

A substantially different problem arises if essentially all data are to be compressed, the data are not known in advance, and the unit for compression is a page or less. A number of studies and designs for this case have proposed new compression techniques as well as system architectures. These include examples from Loughborough University [4], the University of Texas [5], Yonsei University [6], AMD [7], and others. In some cases, compression can be done in software (e.g., [5, 8]); in others, for example [4], a hardware compressor is proposed. A particularly interesting example is the Crusher system [9] developed at IBM Rochester, which may have been the first to have operational hardware.[1] In many respects (e.g., a 1KB unit of compression), system operation was similar to MXT. However, the compression algorithm, the storage and directory architectures, and some aspects of the system structure are different. In comparison with Crusher, MXT offers substantially smaller compression/decompression latency and notably improved compression and storage efficiency.

The compressors used in the above systems and designs are in the class termed compression by textual substitution (e.g., [10]), or, alternatively, Lempel–Ziv compression [11]. Early work on this class of algorithms appears in [12]. These operate by substituting repeated phrases by pointers to other occurrences within the block of data to be compressed, or to a dictionary of such phrases. For cases in which general data are to be compressed, with the unit of compression no greater than 4 KB, constructing a separate dictionary appears inappropriate, and compression operates by replacing phrases with pointers to other occurrences within the block. This requires finding common phrases, a task performed with the aid of hashing (in software), or content-addressable memories (in hardware). The first fully described publication of a compression technique with this property is LZ77 (or LZ1; see for example [11, 13]), of which there are numerous variants. Existing hardware implementations of such variants, such as IBM's ALDC [14] are very fast, compressing one byte per cycle. However, further speedup is critical, especially for decompression. A prominent example of a compressor achieving further speedup is that of X-match, in which, as in LZ77, pointers are substituted for instances of phrases with prior occurrences, but here phrases are extended four bytes at a time, with special-case handling of phrases not aligned on four-byte boundaries. The speedup improves with the length of the data block to be compressed, and is approximately a factor of 1.5 for 4KB pages [4]. This compressor has also been used in other studies [6]. A very similar technique was obtained independently at the University of Texas [5]. A different direction for speeding up coding/decoding was developed in connection with Crusher: The compression window (the part of the compression block examined for earlier examples of the phrase) was limited in length, and the pointer format was restricted for simplicity, so as to permit faster clocking. The result was some speedup, but at the expense of compression efficiency.

Significant speedup over X-match requires that multiple compressors operate simultaneously. One possible approach is to speculatively find matching phrases starting at various points in the block, then choose the appropriate one(s). This unfortunately entails substantial hardware complexity, because essentially an additional search engine is required for each separate speculation. Another possibility is to split the block to be compressed and feed the sub-blocks to separate processors. A problem here is that each sub-block would contain too few repeated phrases. This prompted an investigation of how such multiple compressors could share information so as to obtain good compression efficiency. The result was a family of techniques [15] which yield parallel speedup with little loss of compression. Hardware implementation is discussed in [16]. A member of this family was implemented in MXT, with a level of parallelism of 4.

Even with very rapid decompression, there is still substantial decompression latency for some data. This can be mitigated by maintaining a substantial amount of frequently referenced items in uncompressed form. In some proposals (e.g., [4]), the uncompressed data consist of a set of pages or address space which is accessed as in a standard machine. In one interesting example [7], pages are decompressed on TLB misses. In MXT, the uncompressed data are the contents of a large L3 cache. Cache lines are 1 KB in size, as is the unit of compression, with cache lines decompressed/compressed on access/storeback. **Figure 1** illustrates a block diagram of the system.

In most previous studies, the unit of compression is a 4KB page. This has the advantage that fewer variable-length objects need be stored than for a 1KB unit, but at the expense of greater latency in access and greater required compression/decompression bandwidth. Obtaining the advantage of lower latency and required bandwidth requires an efficient way of storing and indexing the data. Further, it is desirable to avoid the need for reorganization or garbage collection, which may occur if the memory is badly fragmented. The approach taken in MXT [17–19] utilizes fixed-size physical-memory blocks of 256 bytes. This would normally result in excessive fragmentation: A 1KB line compressing into,

[1] J. D. Brown, IBM Rochester, personal communication, 1995.

say, 128 bytes would be allocated 100% more memory than necessary. This condition is avoided via two mechanisms [17–19]: Lines from subgroups such as a page are permitted to share blocks, a strategy termed "roommating" (or "fragment combining"), and lines that compress very well are allocated no blocks, being stored entirely in the directory. The result is a memory organization which wastes little space and permits fast access, with no need for reorganization. **Figure 2** illustrates the memory layout. Virtual pages are allocated "real" addresses, which are actually offsets in a compression-translation table or CTT. Entries in the CTT indicate the location of blocks holding individual 1KB lines.

If the unit of compression is a page, and software is invoked whenever a page is to be decompressed, the operating system (OS) can check whether there is sufficient space before decompression. In other words, the compressed part of memory can be regarded as a more or less standard paging store. A decompressed page is placed in a part of memory which is accessed as in a normal system.

The management of a memory like that of MXT, where individual lines are compressed on access, is rather more complex. Here the set of compressed pages can have varying physical space requirements because of variations in compressibility. If space must be recovered via paging, the act of paging itself can result in additional memory usage due to accessing system page tables and other OS data structures that can expand and/or displace current cache contents. A possible consequence is a system hang, in which the memory controller cannot service a cache fault. We term the ensured avoidance of this condition *guaranteed forward progress*, or GFP. There are various approaches to obtaining GFP. All require reserving some space, which may be needed to ensure the success of pageouts. The space could simply be a set of reserved blocks. A possible alternative would be to have a set of clean (i.e., backed on disk) pages as a reserve, which could be erased when necessary. However, traversing and modifying page tables to find members of this list could itself require substantial reserves (because of possible changes in the compressibility of the page tables and the displacement of existing cache contents during this process). An approach described below is to maintain a list of such pages in a compact data structure we call an *outlist*, which can be processed by the interrupt handler.

Given that the system exhibits GFP, an additional issue is that of efficient virtual memory management [20]. This entails mechanisms for allocating a combination of physical space and real addresses, a task complicated by the fact that the amount of physical space actually available may not be easily observable, but can only be estimated. An approach considered here is to use estimates of the amount of immediately available free space associated with reserves on reclaim lists, together with estimates of quantities such as allocated but unused memory.

The following is a summary of the paper. Section 2 discusses compression algorithms and gives representative results for the compressibility of various workloads. Section 3 discusses the organization of the compressed memory, along with directory structures. It is shown that for a variety of workloads, there is relatively little overhead required over the raw numbers provided by the compressor. Section 4 contains a discussion of compressed-memory management and approaches to the GFP problem. Finally, Section 5 relates these descriptions to the MXT hardware and software described elsewhere in this issue.

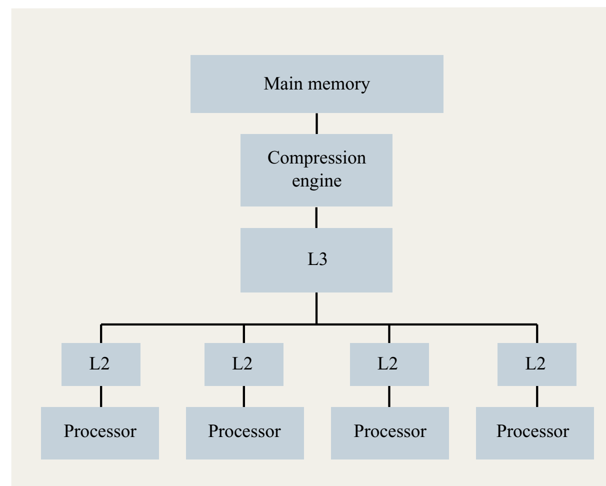

Figure 1

Overall system organization.

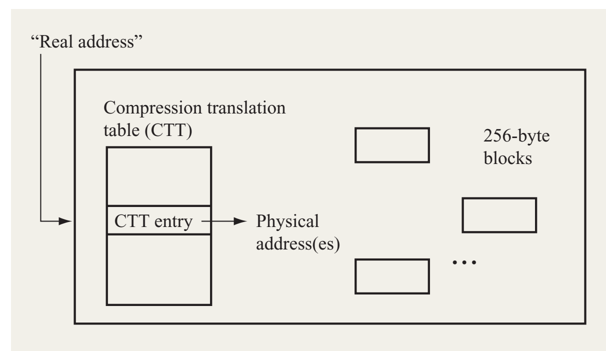

Figure 2

Compressed main memory layout.

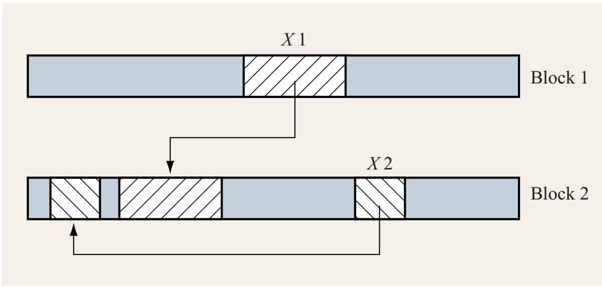

Figure 3

Example of two-way parallel encoding.

2. Compression algorithms

Compression hardware for the memory-compression application must have very low decompression latency and high compression bandwidth. It should operate well on relatively small amounts of data whose characteristics are not known in advance. The only known family of techniques with these properties is compression via textual substitution, in which, as described previously, phrases in the data are replaced by pointers to entries in a dictionary of phrases drawn from the data to be compressed. With small amounts of data, such as 1KB blocks as implemented in MXT, it is inappropriate to construct a separate dictionary. Instead, the dictionary is the data itself. For example, in LZ77 or X-Match, phrases in the data are replaced with pointers to earlier occurrences of each such phrase. Thus, if the block of data to be compressed consists of bytes $A(1), A(2), \cdots, A(n)$, and $A(1), A(2), \cdots, A(r)$, $r < n$, has been compressed at a given point, then in LZ77 the next phrase $A(r+1)$, $A(r+2), \cdots, A(r+q)$ is the longest such phrase previously occurring (and where phrases may overlap). If there is no such phrase with (typically) $q > 1$, then $A(r+1)$ is encoded as a *literal*; that is, the byte $A(r+1)$ is encoded as itself, as indicated (typically) by a flag bit (there are numerous variations; for example, Huffman coding may be used for pointers, phrase lengths, and/or literals; see for example [13]). In the X-Match variation, each $A(i)$ above consists of four bytes of data, and bytes at phrase boundaries are handled as special cases.

Hardware implementation for such encoders can be done using a content-addressable memory, or CAM (for example, see [14]). Referring to the above example, in one CAM access $A(r+1)$ can be compared with $A(1), A(2), \cdots, A(r)$ and all matches found. Assuming that one or more matches have been found, in the next cycle $A(r+2)$ can be compared (again using one CAM access) with $A(1), A(2), \cdots, A(r+1)$, and hardware logic determines whether the next matches found (if any) are consecutive with previous matches. This continues until no further consecutive matches are found, thus yielding all occurrences of the longest matching phrases.

Although the CAM-based hardware implementation is very fast, further speedup is required for the current application. As discussed in the Introduction, X-Match offers some speedup, but only a factor of roughly 1.5 for 4KB pages [4]. For 1KB lines, however, typical phrase lengths are only a few bytes, thus substantially decreasing the speedup of X-Match, since most phrases are now special cases.

A natural approach to speedup is to have multiple compressors operate simultaneously (and similarly, multiple decompressors operating simultaneously for decompression). As mentioned in the Introduction, one method is to speculatively find matching phrases. That is, referring to the above example, one could attempt to find matching phrases starting with the currently uncompressed data beginning with $A(r+1)$, simultaneously with the data beginning with $A(r+2)$, and so on. One problem with this is hardware complexity, since to use a CAM-based implementation, a separate CAM would be needed for each such starting point. Another problem is that the maximum speedup is of the order of the average phrase length, since at most one phrase is produced per cycle.

Another alternative would be to have separate compressors operating on distinct subsets of the data. A problem here is that compression suffers because of the smaller amount of data available to each compressor. In [15, 16] this issue is addressed by introducing a class of LZ-like algorithms in which multiple compressors operate simultaneously on all of the data, subject to constraints that ensure decodability. In practice, a stronger constraint is required, that of single-pass encoding/decoding (the latter being a requirement for low latency). Single-pass encoding/decoding is a property of LZ77, X-Match, and other LZ-like algorithms. However, it is not a necessary condition for compression via textual substitution (see [10, 15]).

Let $B(1), B(2), \cdots, B(n)$ be some ordering of the data $A(1), A(2), \cdots, A(n)$. Suppose we partition this block into k sub-blocks, each of length n/k, and use k compressors to compress (in parallel) each such sub-block, with the condition that each compressor may find matching phrases in *any* of the k sub-blocks, subject to constraints on decodability. As described in [15], the compressed data can be decoded if and only if there are no cycles in the character dependency graph. One way to ensure this is to have a partial order $B(i) < B(j)$ if $1 + [(i-1) \bmod (n/k)] < 1 + [(j-1) \bmod (n/k)]$, and to require that if $B(j)$ depends on $B(i)$, then $B(i) < B(j)$ in the partial order. **Figure 3** illustrates the algorithm for $k = 2$. Here, phrase $X1$ is matched with a phrase from the other block. Phrase $X2$ is matched within the same block.

Figure 4 provides an example configuration of a parallel hardware implementation for the case of a 1KB block split into four 256-byte sections, with four-way parallel CAM-based compression. As shown in the figure, four uncompressed data input streams are routed to separate sections in four replicated CAM-based compressors. Thus, at any point in time, each of the four compressors "sees" not only the previously input data from the stream it is compressing, but the previously input data from the other three streams as well. An alternative hardware design involves using a common memory array for the four CAMs shown (however, four separate comparator arrays are required).

A parameter of interest is the size of the unit of compression. With too small a size, compressibility will suffer. Conversely, too large a size will produce excess decompression latencies and hardware complexity. In [15] tradeoffs between compressibility and sizes of the unit of compression were also investigated; typically there is a knee in the curve at about 512 bytes (an example is shown in **Figure 5**). A unit of 1 KB was chosen as a good compromise.

3. Memory organization

A key difference in memory organization between MXT-like compressed-memory systems and traditional architectures is that there is an additional level of address translation: After virtual-to-real address translation, real addresses are "translated" to physical addresses (in hardware) by means of a CTT (compression translation table). With a 1KB unit of compression, each CTT entry contains either 1) the compressed data itself (for highly compressible data—for example, a 1KB segment consisting of all zeros); 2) pointers to one or more blocks containing the compressed data; or 3) pointers to a number of blocks containing the data in uncompressed format. (A small area of memory, an "uncompressed region," may also be provided in which there is no real-to-physical address translation or compression. That is, real addresses are used directly as physical addresses bypassing the CTT; in MXT such an option is provided at system initialization.)

As an example, with an L3 cache-line size of 1 KB (the unit of compression), and a block size of 256 bytes (as in MXT), after a virtual address is translated, the result is used not only as an associated real-memory address, but also as a pointer or an index to an entry in the CTT. Each CTT entry contains flags, fragment-combining information, and pointers for up to four blocks. At most four blocks are required, since if a given line does not compress, it is stored in an uncompressed format, as indicated by a flag in the directory entry. On an L3 cache miss, the memory controller and decompression hardware find the blocks allocated to store the compressed line and dynamically decompress the line to handle the miss. Similarly, when

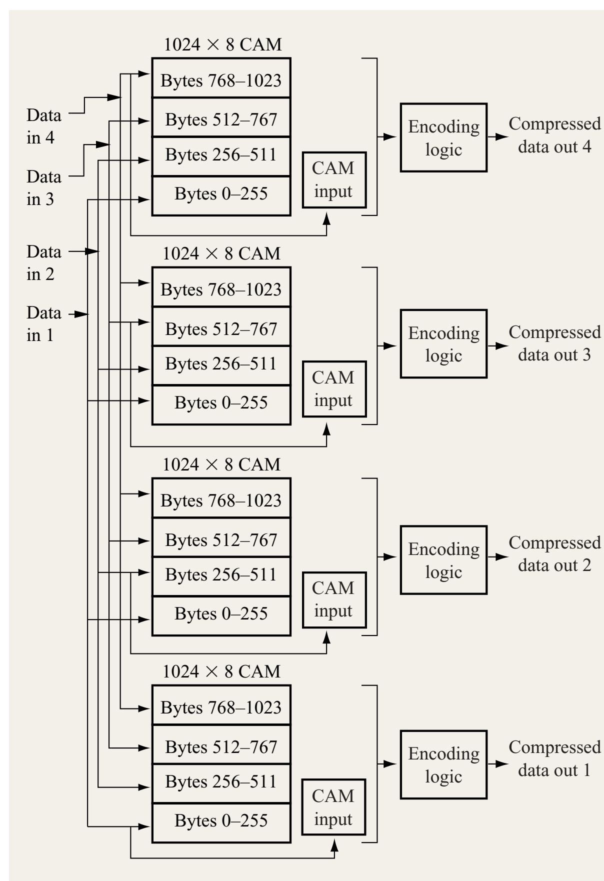

Figure 4

Hardware for four-way parallel compressor.

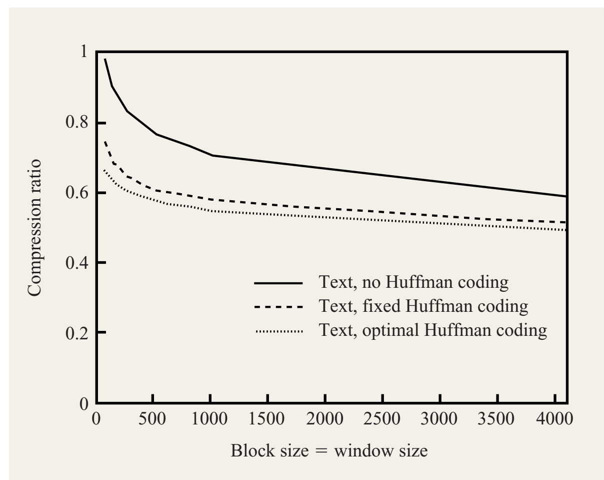

Figure 5

Example of compressibility vs. block size.

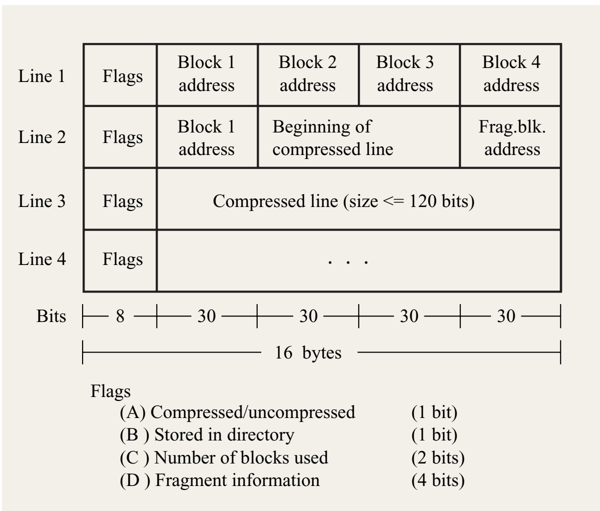

Figure 6

Possible CTT entry formats.

a new or modified line is stored, the blocks currently allocated to the line (if any) are made free, the line is compressed, and then it is stored by allocating the required number of blocks. The last block allocated to a line, called a *fragment*, may be combined and stored in the same block with another fragment from a predetermined set of lines associated with the given line, where this set is called a *cohort*. Since this is done by hardware, under normal operation the fact that main memory is compressed is transparent to the OS. An exception is that because of variations in compressibility, the OS must adapt to changes in the amount of physical memory actually available at any given time (as described in Section 4).

Figure 6 illustrates some possible formats in the general case for the descriptors of four 1KB lines making up a 4KB page in the CTT; a block size of 256 bytes is assumed. For this example, the descriptor for each line has one byte of flag data, and enough space to hold the addresses (30 bits for each address) of four blocks (at most four blocks are required, since this suffices to store the line in an uncompressed format). Note that with 30-bit addresses and 256-byte blocks, this allows 256 GB of addressability. The flags are as follows.

- (A) This flag (one bit) indicates whether the line is stored in an uncompressed or compressed format.
- (B) This bit indicates whether the line is stored entirely within the directory entry (for this example, this applies to lines that compress to 120 bits or less).
- (C) Here, two bits are used to give the number of blocks allocated to store the line (this applies only for lines stored in a compressed format that require more than 120 bits).
- (D) One bit is used to indicate whether the line fragment is at the beginning or end of a shared block (here we assume two-way combining), and three bits are used to store the number of granules (i.e., the smallest unit of storage that can be used to store fragments, which we are assuming is 32 bytes) occupied by the fragment.

The entry for line 1 in Figure 6 has a format which includes the addresses for all of the blocks holding a line. In contrast, line 2 shows an alternate format in which the first data after the flags is a block address, followed by the first 60 bits of the compressed line. This would allow latency in decompression to be overlapped with delays in fetching the next block (at the expense of somewhat greater hardware complexity). In this type of design, each block would contain compressed data followed by a pointer to the next block (if any). The entry for line 3 illustrates the case in which the compressed data are held entirely within the CTT. For MXT, CTT entries analogous to the formats of lines 1 and 3 are used.

There are two general approaches to the design and use of CTTs. In a “static” CTT design, the CTT is allocated (at system initialization) a fixed area of contiguous physical memory of a size sufficient to support a maximum degree of compression. This approach simplifies hardware design, as well as modifications to existing operating systems so as to support compressed-memory systems. From the point of the view of the OS, physical memory is the same size as that provided by the addressability of the CTT, and “exists” at all times. However, if physical memory becomes overcommitted (because of overall decreasing compressibility), the OS can handle this situation by zeroing and removing pages from the list of available page frames (zeroing a page ensures that it can be stored entirely within a CTT entry, thus requiring no additional physical memory). If overall compressibility improves (as indicated by an increasing number of free blocks in the compressed-memory system), these page frames can be added back to the list of available free page frames (see Section 4).

An alternative approach for the design of the CTT is a dynamic directory design. In this approach, the CTT would be noncontiguous, and blocks to hold CTT entries would be allocated and deallocated as required; that is, the logical real memory would grow or shrink dynamically, depending on the overall compressibility of physical memory contents at any point in time. Although this would entail minor additional hardware complexity, it would require significant modifications to existing operating systems so as to support an abstraction of real memory as a system resource that can dynamically vary in

size. The advantage of this approach is that no fixed upper limit on compressibility need be assumed. At the current time, OS use of dynamic CTT designs is an area for further research.

Figure 7 illustrates a static CTT design with cases in which a line is stored entirely within a CTT entry (line 1); a line is compressed and allocated one full block and part of a second block (line 2); and finally two lines in which the fragments of the lines have been combined (lines 3 and 4). A comprehensive study of design alternatives for fragment combining and the effects on overall compressibility is included in [19]. As mentioned above, a cohort is a set of lines that are allowed to share blocks of physical memory. For example, as currently implemented in MXT, cohorts consist of the four 1KB units of compression corresponding to each 4KB page. Some key results of this study are as follows. First, no fragment combining leads to unacceptable overheads (overall compressibility is significantly decreased). However, if fragment combining is used, then among the many alternatives for cohort size (where a cohort, as previously stated, is a predetermined set of lines for which fragment combining can take place), degree of fragment combining (number of fragments that are allowed to share a block), fragment-combining method (e.g., first-fit or best-fit), and cohort determination, very little benefit is obtained beyond using fixed cohorts of size 4 with two-way fragment combining using either first-fit or best-fit. MXT implements this approach: The cohorts consist of the four 1KB lines in each 4KB page, and fragment combining is two-way using best-fit.

It is of interest to see how overall compressibility is affected by the overheads of maintaining a CTT entry for each 1KB line, by fragmentation due to unused space in the last block allocated to each compressed 1KB line (or pair of compressed 1KB lines with combined fragments), and by the effect of choice of block size. **Table 1** illustrates these effects for three memory dumps: "AIX," a memory dump taken from an AIX* system running a memory-intensive logic simulator; "NT(boot)," in which the memory dump took place in a Windows NT** system immediately after the system boot process completed; and "NT(active)," in which the memory dump took place after a number of memory-intensive application programs had been started. The "raw" compression ratio is the average compressed size (in bytes) of all 1KB lines in the memory dump divided by 1024; thus, this represents a bound in the sense that it is the compression that can be obtained when there are no CTT space or storage fragmentation overheads. The "naive" compression ratio is the overall compression including CTT space and storage fragmentation, but in which there is no fragment combining or storage of highly compressible lines within CTT entries: Each 1KB line is stored as some integral number of 256-byte blocks. Finally, the MXT organization compression ratio is the overall compression (that is, including CTT space and storage fragmentation overheads) obtained with 256-byte blocks, cohorts of size 4, two-way fragment combining, and lines compressing to 15 bytes or fewer stored within CTT entries. (The MXT-like organization with 128-byte blocks is similar, except that with the smaller block size, each CTT entry for a 1KB line requires 32 bytes instead of 16, and lines compressing to 31 bytes or fewer can be stored within CTT entries.) The conclusion is that acceptable overheads are achieved using the above approach in MXT. Note that slightly better compression is obtained by using the smaller block size of 128 bytes, which agrees with analytic results (see [19]); however, this doubles the size of CTT entries, which would require some additional hardware complexity.

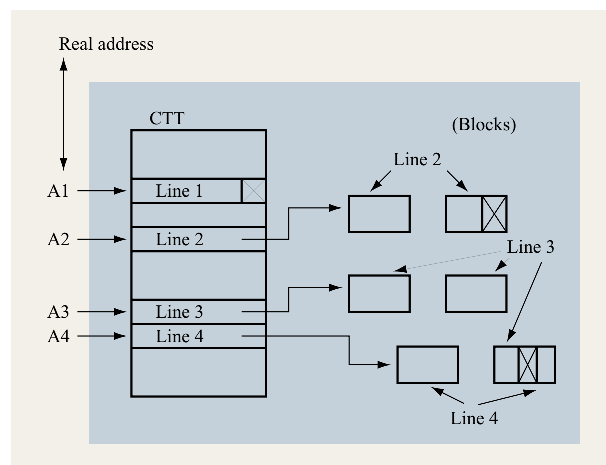

Figure 7

Examples of compressed lines.

4. Operating system support

In this section we consider new issues that an operating system (OS) must confront when supporting a system in which main memory is compressed. Without compression, the OS must manage a memory of fixed size. In such a system there is a one-to-one correspondence between the amount of physical memory and the number of page frames. Because the amount of addressable memory does not change, the number of page frames is generally fixed and known to the OS at boot time. As the number of page frames used by applications changes, the OS can monitor page-frame usage and ensure that there are always enough available page frames to satisfy anticipated requests for new pages. When the number of available page frames drops below some level, the OS performs pageouts, in which pages used by applications (or the OS itself) are

Table 1 Compression of memory dumps.

Memory dump	*Raw compression (%)*	*Naive organization (%)*	*MXT organization (256-byte block) (%)*	*MXT-like (128-byte block) (%)*
AIX	34.1	50.4	42.7	39.7
NT(boot)	48.8	65.2	56.8	54.2
NT(active)	36.3	53.2	44.9	41.7

written to disk (if modified) and then placed in a pool of available page frames.

In contrast, with compression, the OS must manage a memory in which the amount of addressable real memory (corresponding to the number of usable page frames) is variable and changes dynamically according to the compressibility of the pages currently in memory. There is no longer a one-to-one correspondence between the amount of physical memory and the amount of addressable memory used. Both physical memory and addressable memory are resources that must be managed. For example, at boot time the system may be set up with a predefined maximum number of page frames (perhaps based on the expected compression ratio). If compressibility is better than or equal to the expected compression ratio, the OS runs in a traditional way and performs pageouts when running low on available page frames. On the other hand, if compressibility is worse than expected, at some point in time the OS may have more than enough available page frames but be low on physical memory. In this case, the OS should perform pageouts, in which pages are written to disk (if modified) and then cleared so as to free up physical memory. (The zero-page operation implemented in MXT is a useful and fast method for freeing memory.) If compressibility subsequently improves, the OS can allow more page frames to be used by applications, thereby increasing memory utilization.

In this paper we discuss issues related to managing physical memory in two situations: "normal" conditions, in which memory usage changes because of normal fluctuations in either compressibility or the number of used page frames, and extremely low-memory conditions, in which the system is in danger of running out of memory.

Note that the compressibility of memory may change on each L3 castout, and that such castouts are not generally visible to the OS. Therefore, without proper controls, an L3 castout may require more physical memory than is currently available, possibly resulting in a system crash due to an out-of-memory condition. While certain implementations may make such an event highly unlikely, we describe several approaches for ensuring that such an out-of-memory condition can *never* occur, i.e., for ensuring guaranteed forward progress (GFP).

Guaranteed forward progress

Some subtleties which must be considered in designing software that ensures GFP include the following:

- One can never know the exact amount of free physical memory at an instant in time. For example, whereas a register in the MXT memory controller contains a count of the number of free blocks, by the time a read to the register is returned to a processor, the register contents may have already changed.
- Upon access, only the referenced line is decompressed, not the entire page. Thus, previous approaches of permitting access only to uncompressed pages and adjusting the number of compressed pages according to the compression ratio must be modified [4, 7, 8].
- The L3 line size may be different from the unit of compression (compression line). For example, suppose the compression line size is $k = 1$ KB but the L3 line size is $l = 64$ bytes. Then a castout of a single L3 line might cause the entire compression line to expand. A tight upper bound on the expansion is unknown. However, in the worst case each cache line of length l bytes may cause a physical memory expansion no larger than the unit of compression k, for an expansion ratio no greater than (k/l). Thus, a complete L3 cache flush could increase physical memory usage by

$$L = |L3| \times (k/l) = N3 \times k \text{ bytes},$$

where $|L3|$ is the size of the L3 (in bytes) and $N3$ is the number of lines in L3. For example, for a 32MB L3 and the above-mentioned line sizes, a cache flush could expand memory by 512 MB. In MXT, this effect is mitigated, since $k = l = 1$ KB, in which case $L = |L3| = 32$ MB.

We consider two general approaches to ensuring GFP. The first approach limits the number of mapped pages and, roughly speaking, ensures that there is always enough free physical memory available to hold all of the mapped pages even if they expand completely. Since an access to an unmapped page generates a protection fault requiring OS intervention, the OS can decide at the time of the fault whether or not there is enough physical memory to map the additional page. If there is enough space, the

page is mapped and program execution can continue; if not, some other pages must be unmapped before the newly accessed page can be mapped. Pages can be effectively unmapped by setting appropriate protection bits in the page tables. While other variations are possible, in this discussion we assume that when a page is unmapped, its lines are flushed from all caches. Specifically, let F denote the amount of free space (in bytes), let $\boldsymbol{M}$ denote the set of mapped pages (with B_p bytes per page), and let M denote the number of mapped pages. $\boldsymbol{M}$ must always include key OS and driver code, data structures such as those required for paging operations, and all pages involved in I/O. If, when the OS decides whether or not to map a new page, it ensures that

$$F \geq M \times B_\text{p},$$

it is guaranteed that the system will always have enough memory; in the worst case, before the OS gains control, all of the mapped pages could fully expand. Because only mapped pages are in the cache, we need not worry about additional expansion due to a cache flush. A method for ensuring that this relationship always holds is the subject of a pending U.S. patent application [21]; that method includes handling the problems that F is continually changing and there are delays in flushing lines from the cache. The above free-space bound can be improved somewhat by keeping track of the number of mapped read-only pages. Compared to the approach of [4], in which entire pages are decompressed, this approach has the advantage of decompressing only accessed lines, which both reduces the utilization of the compression/decompression hardware and results in lower-latency decompression times. Another advantage of this approach is that it is free of interrupts. However, a major disadvantage is that if M is small, the mapping/unmapping overhead may be excessive, while if M is large, the free-space requirement is excessive. In addition, this approach may severely limit the number of pages that can be pinned at any one time (to less than M if the system requires pinned pages to be mapped).

A second general approach assumes that a low level of available physical memory can be detected, for example by having the memory controller signal an interrupt to the processors when free space becomes too low, and that this interrupt causes the processors to stop all processing except for that which is necessary to free memory (and incoming I/O). We now examine this approach in more detail. Suppose that the physical memory consists of P pages, i.e., that the memory can hold only P uncompressed pages. If, under compression, the system has more than P pages, the system is exposed to a potential out-of-memory condition. For GFP, the system must be able, through a series of steps, to reduce the number of pages in use to P. (This places a limit on the number of pinned pages.) The difficulty is that, while performing pageouts, physical memory usage might actually *increase* for a period of time due to L3 castouts, and the fact the system data structures, such as page tables, might become less compressible as they are modified during the pageout process. For GFP, the memory controller must signal the interrupt when there is still enough free memory so that the system does not run out of memory, even in the worst-case expansion during pageouts.

Suppose the controller signals a low-memory interrupt when the amount of free physical memory drops below some extreme low-memory threshold $T1$. Upon receipt of this interrupt, all "nonessential" processing is halted as soon as possible. (By nonessential, we mean any activity that is unrelated to either incoming I/O or the freeing of physical memory, which might already be in progress since the system may have observed, or been alerted to, a decrease in free memory prior to the extreme low-memory threshold.) The OS then clears pages until the amount of free memory exceeds some level $T2$ where $T2 > T1$. When this "safe" level $T2$ is reached, the system is put in a state where it can recover again if memory drops below $T1$. Then normal processing can resume. If $T1$ is chosen large enough so that the safe level $T2$ can always be reached, the system has GFP. To analyze the required level, let $F1$ denote the worst-case expansion until the low-memory interrupt is recognized by the OS. Let $F2$ denote the worst-case expansion from the time the interrupt is recognized on all processors until all nonessential processing is halted and the required number of pageouts are performed. We then require that

$$T1 \geq F1 + F2.$$

If the above equation is satisfied, GFP is ensured, since there is always enough memory to achieve an increase in free memory (this is the subject of a pending U.S. patent application [22]). We now consider the terms $F1$ and $F2$ in more detail. $F1$ is the expansion possible until the interrupt is recognized on all processors. When the interrupt is generated, there is a small (worst-case) time delay, $t1$ (seconds), until it is received at the processors. Upon receipt, if interrupts are not disabled, an interrupt routine is entered (all activity up to this instant is included in $F1$). However, if the OS is executing a critical piece of code, it may be running with interrupts disabled. Let $t2$ denote the maximum length of time the OS can run with interrupts disabled. Thus, the interrupt is recognized by time $(t1 + t2)$, and if the maximum number of L3 writebacks is w per second,

$$F1 \leq (t1 + t2) \times w \times k \text{ bytes}.$$

Notice that the factor k, the compression line size, is used in this equation rather than l, the L3 line size; each L3

writeback of l bytes might cause memory utilization to increase by k bytes.

Now consider $F2$, the time to stop all nonessential processing and perform pageouts. (One way to accomplish this is for the OS to dispatch "null tasks" on all but one of the processors in an SMP; pageouts are performed on the reserved processor.) We decompose $F2$ into the following three components:

- I, the space required for all I/O operations.
- $F3$, the worst-case expansion due to changes in compressibility of system data structures (such as page tables) during processing in this phase.
- C, the worst-case expansion due to cache-flush effects.

We thus write

$$F2 = I + F3 + C.$$

First, consider I, the space required for I/O. Suppose the OS has permitted I/O for p pages to commence. These pages may occupy no space in physical memory (except for the directory entry) and so at least $p \times B_p$ bytes must be reserved. Thus,

$$I \le p \times B_p.$$

Now consider $F3$. Suppose the total number of distinct L3 lines (the "footprint") of the software that can execute in this phase is $n2$, but only $n3$ lines of these can be modified. Such lines might consist of page-table entries, dispatcher data structures, or data structures in I/O drivers. Since the current compressibility of these $n3$ lines is unknown, in the worst case they could occupy no space in physical memory and expand completely during processing. Thus,

$$F3 \le n3 \times k.$$

Finally, consider C, the maximum expansion due to cache-flush effects. Each of the above $n2$ lines may displace a line that is already in the cache. In addition, if I/O flows through the L3 cache, I/O reads and writes may displace lines already in the cache. Suppose that the p pages for I/O represent $n1$ L3 lines; then $(n1 + n2)$ lines may be displaced from the L3 cache. However, if the total L3 size is $N3$ lines, the cache-flush effect is limited to at most $N3$ lines [since the second time a physical line in the cache is flushed, it is at worst writing back one of the $(n1 + n2)$ lines for which worst-case expansion space has already been reserved]. Thus,

$$C \le \min\,[N3, (n1 + n2)] \times k.$$

Combining the above upper bounds on $F1$, $F2$, $F3$, I, and C yields a lower bound on the threshold $T1$ required for GFP. Note that this bound depends on a variety of hardware- and software-related quantities, including the maximum time the OS can run with interrupts disabled and the maximum footprint of the pageout software. In particular, the term $n2$ might be quite large. For example, in Microsoft Windows NT (see [23]), the usual pageout code involves having the OS "trim" working sets, which is accomplished by scanning the page tables of processes, changing page-table entries (resulting in possible expansion), and executing pageout code. To improve on such a large factor, it is desirable to implement a special low-footprint data structure, called an *outlist*, consisting of clean pages (i.e., pages for which a valid copy exists on disk) which can easily be zeroed (this is the subject of a pending U.S. patent application [20]).

The outlist is a software construct allowing physical space recovery by an interrupt handler or by a service processor, outside the context of OS virtual-memory management. Pages can be invalidated and cleared without the usual locks on page tables and page-frame databases. This approach reduces the amount of physical space that must be reserved to permit working set trimming and pageouts. The outlist is a list of clean pages that can immediately be cleared, avoiding the need for a general traversal of page tables and paging I/O. Every page on the outlist is marked invalid, so that it cannot be accessed or modified without OS intervention. We may thus think of the outlist as representing a subset of a traditional "reclaim" list (the "standby" list in Windows NT). Normal maintenance of the outlist (when there is no space shortage) requires a lock. However, when space is an issue, the list may be traversed and pages cleared without locking. The pages on the outlist must contain adequate physical storage to enable the OS to trim, page out, and return to normal operation; i.e., once the pages on the outlist are cleared, the OS has enough space to run the usual paging code to increase the number of clean pages in the system, and to replenish the outlist with clean pages before returning to normal operation. The outlist itself could be organized as a hash table, keeping its memory footprint small and its processing overhead at a minimum. The effect of this approach is that the terms $n2$ and $n3$, which contribute to $F3$ and C, can be reduced; i.e., the system can have GFP and run with lower free-space reserves.

An outlist entry contains the page number, a count of physical space used, and a presence flag. Normal outlist maintenance is done within the context of the OS virtual-memory manager; the state of pages in the outlist is kept consistent with the page tables and page-frame database. Atomically updating the page presence flag provides coordination between an interrupt handler recovering space and outlist maintenance code in the OS.

There are three types of operations on the outlist:

- Normal: additions and deletions of pages done in the context of the virtual memory manager.
- Flush: clearing of pages on the outlist by the interrupt handler or service processor to recover physical space.
- Sweep: before resuming normal operation, updating of page tables and page-frame database to reflect flush operations.

The total physical space consumed by the pages of the outlist, $F0$, is atomically adjusted as pages are added, removed, or cleared. If $F0$ drops below the minimum required to run the paging manager, new pages must be added to the outlist. Notice that this is significantly different than keeping the amount of physical free space at least equal to the paging manager's footprint, since the pages on the outlist can still be accessed (after a software interrupt).

Even with the outlist, GFP depends on having some information about the maximum footprint of certain software code paths, which might be difficult to obtain in practice. One way to reduce this uncertainty is to run the kernel (or key parts of it, such as the outlist) uncompressed (this is the subject of a pending U.S. patent application [24]). In this case, OS structures cannot expand.

Normal operations

In addition to handling low-memory situations, the OS must also manage physical memory efficiently as the compression ratio changes. Essentially, the OS must be able to reduce the number of used pages in the system when free space begins to become low, and increase the number of pages when it is plentiful.

We describe several issues associated with such memory-management approaches, and in particular address the question of how much memory the OS should regard as "available." The naive answer is to simply read the register indicating the current amount of free memory; if the measurement is taken at time t, call this number $F(t)$. While the OS could make decisions based solely on $F(t)$, some improvements are possible.

First, suppose the system keeps track of the amount of space represented by pages on the reclaim list, $R(t)$. As its name suggests, this space can easily be freed, so the OS could consider such space as available and base decisions on both the actual level of free space, $F(t)$, and an estimate of the amount of available space $A(t) = F(t) + R(t)$. That is, in addition to a threshold-control policy on just $F(t)$, a parallel, cooperating threshold-control policy on $A(t)$ can be implemented. For example, if $A(t)$ exceeds some threshold [and $F(t)$ is not too low], the OS can satisfy a request for a new page by supplying a page-frame ID from either the free or the zeroed-page lists (if any). As $A(t)$ drops below some other threshold, new page requests can be satisfied by taking the page-frame ID off the reclaim list; on average, this keeps the compression ratio the same. If $A(t)$ continues to drop, $A(t)$ can be increased by building up the reclaim list (through pageouts). This parallels traditional OS policies that track and control the number of page frames in use. It further helps to ensure an adequate supply of easily freed space as $F(t)$ decreases, thereby reducing the chances of a low-memory interrupt.

In MXT, a potential source of rapid memory expansion occurs when the OS has recently given out many new pages from the zeroed-page list. When first given out, such a page occupies no space in physical memory. In fact, immediately after such a page is given out, the estimated compression ratio actually improves, since the amount of memory used is unchanged but the number of used page frames increases. It is only as the lines from such a new page age out of the L3 that they begin to occupy space in memory. If there are many such pages, a naive policy would conclude that even more pages can be given out, since the average compression ratio has improved. Because of the time delays involved, it is therefore possible to overcommit memory if many new pages have recently been given out. One way to counteract this situation is described in [25], in which the notion of "allocated but unused" storage is introduced. Here, the estimated amount of available storage is given by $A(t) = F(t) + R(t) - v(t)$, where $v(t)$ is an estimate of the allocated but unused storage. If many pages have recently been given out, $v(t)$ tends to be large, and the system's estimate of available space is reduced (thereby reducing the chances of overcommitting memory). A dynamic method of estimating $v(t)$ is described in [25]; each new page grant initially increases $v(t)$, representing increased potential memory usage while the page's lines are still in the cache (and not reflected in physical memory). This increase in potential memory usage is decreased over time as the page's lines are expected to age out of the cache (and be reflected in physical memory). This method is efficient, and simulation studies have indicated that it is reasonably accurate.

Compressibility is expected (and has been observed) to vary as the workload changes, e.g., when loading new applications. However, for certain long-running workloads, it has been observed that compressibility remains quite stable; Reference [26] describes the dynamic behavior of a logic-simulation application running under MXT. Thus, while appropriate OS controls on compression are required, for stable workloads we expect those controls to be essentially inactive most of the time. In the Appendix,

we describe several stochastic models that help to explain this behavior and give a qualitative description of the types of fluctuations expected to be observed.

Finally, we briefly describe current software written for MXT [26]. These include a Linux in-kernel modification and an out-of-kernel Windows NT (and Windows** 2000) driver. In both cases, the amount of available physical memory is monitored and actions taken when compressibility changes. The amount of free space held in reserve is determined heuristically; there is no guarantee of forward progress. Of particular interest is the Windows 2000 driver. The driver keeps a number of pinned and zeroed pages in reserve. (Pinned pages cannot be removed by the OS, and zeroed pages occupy no physical space in MXT.) If physical memory is plentiful, the driver returns some of these pages to the OS for use by other programs. If free space becomes scarce, the driver obtains, zeros, and pins additional pages. Experiments indicate that these approaches work well, but this is an area requiring additional research and experimentation.

5. Summary and conclusions

The IBM MXT compression/decompression hardware permits the size of main memory to be effectively doubled with little or no performance impact by storing most of main memory in compressed format. To conceal the performance penalty associated with compression and decompression, compressed memory is hidden behind a large L3 cache. The hardware compresses a line on writeback from the L3 and decompresses the line on access before loading it into the L3. Efficient support of such a system requires new algorithms, machine organizations, and data structures. This paper has provided an overview of several key advances that have enabled development of the efficient MXT hardware and given insight into effective OS support of compressed-memory machines. Among these advances are the following:

1. Fast parallel compression/decompression algorithms that are suitable for implementation in hardware. In MXT, the degree of parallelism for both compression and decompression is 4.
2. Efficient methods of addressing and storing compressed data in memory. A compression translation table (CTT) in the memory controller provides a real-to-physical address translation that enables application programs to run unmodified; i.e., compression is completely transparent to all applications. The CTT is organized in such a way that fragmentation of memory is minimal and garbage collection is unnecessary. In MXT, the compression line size is 1 KB, equal to the L3 line size, and compressed memory is stored in 256-byte blocks. Fragments from two different lines in the same 4KB page can share a segment (two-way combining with a cohort size of 4). Lines compressing to less than 120 bits are stored in the CTT itself.
3. Several software-memory-management approaches which guarantee that the system does not encounter an out-of-memory condition due to rapid changes in compressibility. These approaches require that a certain quantifiable amount of free memory be maintained in reserve. A new data structure, the outlist, reduces this reserve requirement. While these approaches have not been directly implemented in the system-support software for MXT, the principles underlying guaranteed forward progress (GFP) have been influential in the design of this software.

We continue to investigate issues related to main-memory compression and to develop improvements over the current MXT implementation. For example, a machine organization in which the amount of memory stored in uncompressed format is variable and changes dynamically with the compression ratio is described in [27]. In such a system, if compression is better than expected, more lines can be stored uncompressed, thereby further reducing decompression latency.

Appendix: Stochastic models of compressibility

First, suppose that the memory consists of N lines and consider the compression ratio at two distinct times. Let X_n be the change in the compression ratio of the nth line during the time interval. Then the change in compression ratio is $\bar{X}_N = (X_1 + \cdots + X_N)/N$. We model $\{X_n\}$ as a random process. If the workload is stable, then, on average, compressibility does not change, so $E[X_n] = 0$. In addition, under broad assumptions on stationary processes which basically state that the correlation between X_n and X_{n+k} drops off rapidly enough for large k (e.g., if $\{X_n\}$ is a Markov chain), $\bar{X}_N$ obeys a central-limit theorem for large N; that is, $\bar{X}_N$ is approximately normally distributed with mean 0 and standard deviation $\sigma/\sqrt{N}$ for some constant σ which includes the effect of correlation (see p. 375 of [28]). Thus, we expect the change in compression ratio to be of order $1/\sqrt{N}$ with high probability. In fact, under somewhat stronger conditions a "large deviations" principle holds, which states that for any constant $a > 0$ representing a change in compression ratio, there is a constant t (depending on a) such that

$$P\{|\bar{X}_N| > a\} \leq \exp\{-tN + O(N)\};$$

i.e., the probability of a large change in compression ratio is exponentially small (see p. 15 of [29]).

A second model of compressibility relies on statistical results on sampling from a finite population (see [30]). Consider a system such as a database consisting of M pages and suppose that the compression ratio of these pages is c. Now suppose that the main memory contains

some number $N < M$ of these pages, and let $f = N/M$ be the fraction of the database that is in memory. Let c_N be the compression ratio of the pages in memory. If the pages in memory represent a random sample of the entire database, then c_N has mean c and variance $(1 - f)S^2/N$, where S^2 is the sample variance of the compression ratio of all of the pages in the database. In addition, for large M, c_N is approximately normally distributed with the above mean and variance. Thus, the compression ratio of the memory will again have fluctuations about c which are of order $1/\sqrt{N}$ with high probability.

Acknowledgments

The authors wish to acknowledge valuable interactions with C. Benveniste, J. D. Brown, M. Hack, C. O. Schulz, T. B. Smith, R. B. Tremaine, and M. Wazlowski. Jeff Brown first brought the topic of compressed-memory systems to our attention; Michel Hack and Charles Schulz collaborated on adapting our results on memory organization to the MXT architecture described elsewhere in this issue. Michel Hack, Charles Schulz, and Basil Smith participated in many lively discussions on operating system questions. Their contributions are partially described elsewhere in this issue, and in Reference [21]. Brett Tremaine and Michael Wazlowski provided valuable grounding on questions of hardware practicality, as well as data used, for example, in Reference [25].

*Trademark or registered trademark of International Business Machines Corporation.
**Trademark or registered trademark of Microsoft Corporation.

References

1. R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "IBM Memory Expansion Technology (MXT)," *IBM J. Res. & Dev.* **45,** No. 2, 271–285 (2001, this issue).
2. *A Technical Guide to ESA/390 Compression*, Document No. GG24-4130-00, IBM International Technical Support Center, Poughkeepsie, NY, April 1994.
3. T. M. Kemp, R. K. Montoye, J. D. Harper, J. D. Palmer, and D. J. Auerbach, "A Decompression Core for PowerPC," *IBM J. Res. & Dev.* **42,** No. 6, 807–812 (1998).
4. M. Kjelso, M. Gooch, and S. Jones, "Performance Evaluation of Computer Architectures with Main Memory Data Compression," *J. Syst. Arch.* **45,** 571–590 (1999).
5. S. F. Kaplan, "Compressed Caching and Modern Virtual Memory Simulation," Ph.D. Thesis, University of Texas at Austin, December 1999.
6. J.-S. Lee, W.-K. Hong, and S.-D. Kim, "Design and Evaluation of a Selective Compressed Memory System," *Proceedings of the International Conference on Computer Design*, IEEE, 1999, pp. 184–191.
7. J. R. MacDonald, D. Dutton, and S. Cox, "Memory Paging System and Method Including Compressed Page Mapping Hierarchy," U.S. Patent 5,696,927, December 9, 1997.
8. F. Douglis, "The Compression Cache: Using On Line Compression to Extend Physical Memory," *Proceedings of the Winter 1993 USENIX Conference*, USENIX Association, San Diego, 1993, pp. 519–529.
9. W. P. Hovis, K. H. Haselhorst, S. W. Kerchberger, J. D. Brown, and D. A. Luick, "Compression Architecture for System Memory Applications," U.S. Patent 5,812,817, September 22, 1998.
10. J. Storer and T. Szymanski, "Data Compression via Textual Substitution," *J. ACM* **29,** No. 4, 928–951 (1982).
11. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Info. Theory* **IT-23,** No. 3, 337–343 (1977).
12. B. Tunstall, "Synthesis of Noiseless Compression Codes," Ph.D. Thesis, Georgia Institute of Technology, Atlanta, September 1967.
13. T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.
14. D. Craft, "A Fast Hardware Data Compression Algorithm and Some Algorithmic Extensions," *IBM J. Res. & Dev.* **42,** No. 6, 733–746 (1998).
15. P. Franaszek, J. Robinson, and J. Thomas, "Parallel Compression with Cooperative Dictionary Construction," *Proceedings of the DCC '96 Data Compression Conference*, IEEE, 1996, pp. 200–209.
16. P. Franaszek, J. Robinson, and J. Thomas, "Parallel Compression and Decompression Using a Cooperative Dictionary," U.S. Patent 5,729,228, March 17, 1998.
17. P. A. Franaszek, "System and Method for Reducing Memory Fragmentation by Assigning Remainders to Share Memory Blocks on a Best Fit Basis," U.S. Patent 5,761,536, June 2, 1998.
18. P. A. Franaszek, "A System and Method of Compression and Decompression Using Store Addressing," U.S. Patent 5,864,859, January 26, 1999.
19. P. A. Franaszek and J. T. Robinson, "On Internal Organization in Compressed Random-Access Memories," *IBM J. Res. & Dev.* **45,** No. 2, 259–270 (2001, this issue).
20. P. A. Franaszek and D. E. Poff, "Reclaim Space Reserve for a Compressed Memory System," IBM patent application, August 2000.
21. P. A. Franaszek, M. Hack, C. S. Schulz, and T. B. Smith, "Space Management in Compressed Main Memory," IBM patent application, August 1996.
22. P. A. Franaszek and P. Heidelberger, "Compression Store Free-Space Management," IBM patent application, February 1998.
23. D. A. Solomon, *Inside Windows NT*, Second Edition, Microsoft Press, Redmond, WA, 1998.
24. P. A. Franaszek, P. Heidelberger, and D. E. Poff, "Kernel Identification for Space Management in Compressed Memory Systems," IBM patent application, November 1998.
25. P. Franaszek, P. Heidelberger, and M. Wazlowski, "On Management of Free Space in Compressed Memory Systems," *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, ACM, 1999, pp. 113–121.
26. B. Abali, H. Franke, D. E. Poff, R. A. Saccone, Jr., C. O. Schulz, L. M. Herger, and T. B. Smith, "Memory Expansion Technology (MXT): Software Support and Performance," *IBM J. Res. & Dev.* **45,** No. 2, 287–302 (2001, this issue).
27. C. Benveniste, P. Franaszek, and J. Robinson, "Cache-Memory Interfaces in Compressed Memory Systems," *Research Report RC-21662*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, February 4, 2000.
28. P. Billingsley, *Probability and Measure*, Second Edition, John Wiley & Sons, Inc., New York, 1986.
29. J. A. Bucklew, *Large Deviations Techniques in Decision, Simulation, and Estimation*, John Wiley & Sons, Inc., New York, 1990.

30. W. G. Cochran, *Sampling Techniques*, Third Edition, John Wiley & Sons, Inc., New York, 1977.

Received September 13, 2000; accepted for publication March 26, 2001

Peter A. Franaszek *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (paf@us.ibm.com).* Dr. Franaszek received the Ph.D. degree in electrical engineering from Princeton University in 1965. From 1965 to 1968, he was employed by Bell Laboratories. He joined the IBM Research Division in 1968. During the academic year 1973–1974, he was on sabbatical leave at Stanford University as Consulting Associate Professor of Computer Science and Electrical Engineering. He is currently Manager of Systems Theory and Analysis. His interests are in the general area of information representation and management, and computer system organization. Dr. Franaszek has received two IBM Corporate Awards for his work on codes for magnetic recording, an IBM Corporate Patent Portfolio award for his contribution to the ESCON architecture, and Outstanding Innovation Awards for fragmentation-reduction algorithms, network theory, concurrency-control algorithms, run-length-limited codes, and the code used in ESCON, Fiber Channel, and Gigabit Ethernet. He is a member of the IBM Academy of Technology and a Master Inventor. He is a Fellow of the IEEE, and received the 1989 Emmanuel R. Piore Award from the IEEE for his contributions to the theory and practice of constrained channel coding in digital recording. Dr. Franaszek holds thirty-six patents and has published more than forty technical papers.

Philip Heidelberger *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (philiph@us.ibm.com).* Dr. Heidelberger received a B.A. degree in mathematics from Oberlin College in 1974 and a Ph.D. degree in operations research from Stanford University in 1978. He has been a Research Staff Member at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York, since 1978. His research interests include modeling and analysis of computer performance, probabilistic aspects of discrete-event simulations, and parallel simulation. He has won Best Paper awards at the ACM SIGMETRICS and ACM PADS (Parallel and Distributed Simulation) Conferences, and he was twice awarded the INFORMS College on Simulation's Outstanding Publication Award. Dr. Heidelberger has recently served as Editor-in-Chief of the ACM's *Transactions on Modeling and Computer Simulation*. He is the General Chairman of the ACM SIGMETRICS/Performance 2001 Conference and has served as the Program Chairman of the 1989 Winter Simulation Conference and as the Program Co-Chairman of the ACM SIGMETRICS/Performance '92 Conference. He is a Fellow of the ACM and the IEEE.

Dan E. Poff *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (poff@us.ibm.com).* Mr. Poff is a Systems Programmer at the IBM Thomas J. Watson Research Center, where he designs and develops MXT software compression controls. Before joining the Research Center in 1982, he programmed logic chip testers at IBM in East Fishkill, New York. At the Watson Research Center, he first joined a group that developed IBM's first port of UNIX to the first RISC machine, then assisted in porting Carnegie Mellon University's MACH to an early SMP RISC machine. He subsequently assisted in porting MACH to RS/6000. In the early 1990s he joined a group porting Windows NT to the IBM PowerPC. He has received an IBM Outstanding Technical Achievement Award. Mr. Poff received an M.A. degree in history and philosophy of science from Indiana University in 1969 and a B.S. degree in physics from the University of Cincinnati in 1964. He has five patents pending and several publications, and he is a member of the ACM.

John T. Robinson *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (robnson@us.ibm.com; http://www.research.ibm.com/people/r/robnson/).* Dr. Robinson received the B.S. degree in mathematics from Stanford University in 1974, and the Ph.D. degree in computer science from Carnegie Mellon University in 1982. Since 1981, he has been with the IBM Thomas J. Watson Research Center, Yorktown Heights, New York. His research interests include database systems, operating systems, parallel and distributed processing, design and analysis of algorithms, and hardware design and verification. He is a member of the ACM and the IEEE Computer Society.