On internal organization in compressed random-access memories

by P. A. Franaszek
J. T. Robinson

The design of a compressed random-access memory (C-RAM) is considered. Using a C-RAM at the lowest level of a system's main-memory hierarchy, cache lines are stored in a compressed format and dynamically decompressed to handle cache misses at the next higher level of memory. The requirement that compression/decompression, address translation, and memory management be performed by hardware has implications for the directory structure and storage allocation designs used within the C-RAM. Various new approaches, summarized here, are necessary in these areas in order to have methods that are amenable to hardware implementation. Furthermore, there are numerous design issues for the directory and storage management architectures. We consider a number of these issues, and present the results of evaluations of various approaches using analytic methods and simulations. This research was done as part of a project to explore the feasibility of compressed-memory systems; it forms the basis for the memory organization of IBM Memory Expansion Technology (MXT).

1. Introduction

With higher processor speeds but a lack of a corresponding speedup in disk access, the tendency in server-class computers is toward increasingly large main-memory size in proportion to processor speed. A result is the expectation that memory will be a dominant cost factor in the central electronic complex, comprising in the near future possibly fifty to ninety percent of the cost of typical large machines, despite the usual trends in decreasing memory cost. A possible approach to mitigating this cost is to compress the contents of main memory. It is well known that such contents are generally compressible by a factor of 2 or better, and some commercial programs have been available to exploit this fact (for example, see [1]). Such observations have led to the consideration of systems in which the contents of main memory are maintained in compressed form, and decompressed/compressed on a page basis (for example, see [2, 3]). However, when the unit of compression is a cache line (that is, cache lines in main memory are decompressed on misses and recompressed on writebacks), it is necessary to develop some new technology in order to obtain a practical system. Here we describe results associated with the development of IBM Memory Expansion Technology (MXT) [4]. These results form the basis for the design of the compressed-memory organization of MXT (for additional details see [5–7]).

0018-8646/01/$5.00 © 2001 IBM

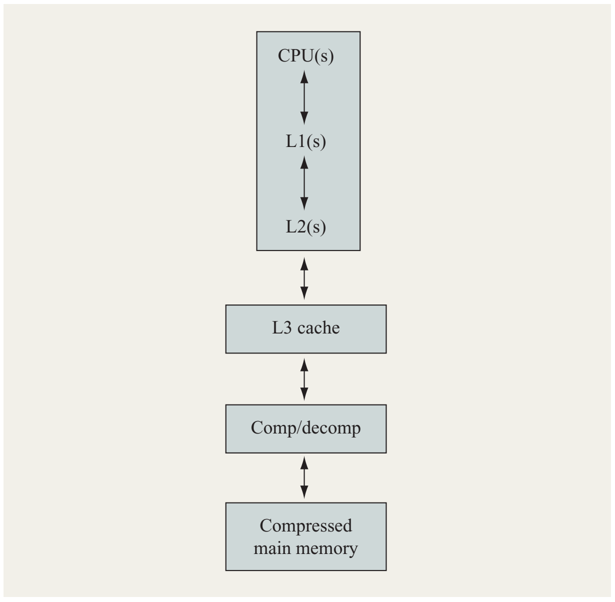

Figure 1

System structure overview.

First, very fast compression/decompression hardware is required, permitting operation at main-memory bandwidths. There has been some progress in this direction, based on extensions to Lempel–Ziv (LZ) methods. In particular, parallel, shared-dictionary techniques [8], implemented using content-addressable memory (CAM), permit effective compression/decompression at main-memory bandwidths. A serial approach, but based on a larger alphabet of multiple byte entries [2], also offers some speedup over standard LZ approaches.

Next, changes in memory management must be made to the operating system, since with compression, the logical total main-memory size may vary dynamically. Issues associated with such management are discussed in [9]. Finally, a way must be found to efficiently store and access the variable-length objects obtained from compression. This problem is the main topic of the current paper. We consider the design of a compressed random-access memory (C-RAM) with the following properties:

1. Logically, the memory M consists of a collection of randomly accessible fixed-size lines, where L is the line size.
2. Internally, the ith line is stored in a compressed format as $L(i)$ bytes, where $L(i) \leq L$, and where $L(i)$ may change on each cache cast-out of this line. $L(i) \leq L$ is ensured by the capability of storing lines in an uncompressed format with suitable directory entries.
3. M comprises a standard random-access memory with a minimum access size (granule) of g bytes. We will generally assume that g is 32.
4. Memory accesses invoke a translation between a logical line address and an internal address. This correspondence is stored in a directory D contained in M. Translation, fetching, and memory management within the C-RAM are carried out by a memory controller rather than by operating system (OS) software.

Use of the directory D to access memory generally implies an added level of addressing indirection, requiring an additional memory fetch. Also, the compressed format may require some average additional latency to handle misses from the next higher level in the memory hierarchy. As described below, this latency can be traded off against compression by varying the line size; sizes in the range of 512 to 1024 bytes appear suitable. Line sizes of this magnitude may require a third level of caching (L3), one more than is typical for many of today's systems. However, L3 caches may be necessary in any case (even without compression), since it is difficult to bring substantially larger memories uniformly close to the processors. The result is that with typical anticipated L3 miss ratios, we expect that average memory access latency will not be significantly affected by the use of a C-RAM for main memory. In the following, we generally assume an L3 line size L of 1024 bytes. The value for L was chosen via a combination of compression results using shared-dictionary parallel compression [7] and estimated L3 cache-miss latencies. An overview of the system structure, showing the L3 cache and the compressed main memory, is given in **Figure 1**.

Three classes of techniques for managing a memory of the above type are 1) organizing M to be a linear space, where variable-length intervals are allocated and deallocated; 2) organizing M as a collection of blocks of possibly multiple sizes, where space for a variable-length object is allocated as an integral number of such blocks; and 3) organizing M as a collection of blocks, but permitting a variable amount of space to be allocated within a block. The techniques we investigate here are of the third type. We use blocks of a single fixed size, and compressed lines are allocated some integral number of blocks, with leftover pieces (termed fragments) from possibly multiple lines sharing an additional block. A line fragment shares a block with line fragments drawn from a "cohort" of other lines. The smaller the cohorts, the fewer the memory operations required to store a line, but the larger the potential fragmentation. This is also true for the number of line fragments permitted to share a block. A principal observation is that the size of the cohorts and the number of lines allowed to share a block can be quite small, with low resulting fragmentation. The result is that the technique is of low complexity, and is suitable for implementation in hardware.

2. Design overview

A high-level design for a system using a C-RAM for the lowest level of the main-memory hierarchy is shown in **Figure 2**. As discussed in the Introduction, we assume that the C-RAM memory M is used to read and write lines resulting from cache misses and stores, respectively, at the next higher level of memory in the memory hierarchy, shown as L3 in Figure 2. The C-RAM memory consists of two parts, the directory D and a collection of fixed-size blocks. Highly compressible lines may be stored entirely within directory entries. Otherwise, the directory entry points to one or more of the fixed-size blocks, which are used to store the line in its compressed format. In the case that fragments from two or more lines are combined and stored in a common block, the directory entries corresponding to the given lines also contain the information necessary to find the fragments.

As usual, virtual addresses are translated to real addresses by means of page tables, which can be of various types depending on details of the processor architecture and operating system. Here, a "real" address corresponds to the location in D of the directory entry for the line.

As an example, suppose that pages are of size 4 KB, and that the L3 cache immediately above the C-RAM has a line size of 1 KB. In contrast to the usual type of page-table design, when 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 C-RAM directory D. Compressed lines that do not fit entirely within directory entries are stored using one or more fixed-size blocks, which for this example we assume to be of size 256 bytes (the issue of choosing an optimal block size is discussed in Section 5). Each directory entry contains flags, fragment-combining information, and pointers for up to four blocks. At most four blocks are required, since we assume that if a given line does not compress (in general, this is always a possibility given a fixed compression method), it is stored in an uncompressed format, as indicated by a flag in the directory entry. Finally, sufficiently compressible lines are stored in the directory entry itself (as indicated by another flag bit), in which case no blocks need be allocated.

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 a new or modified line is stored, the blocks currently allocated to the line are made free (if the line currently resides in the C-RAM), and the line is then compressed and stored in the C-RAM

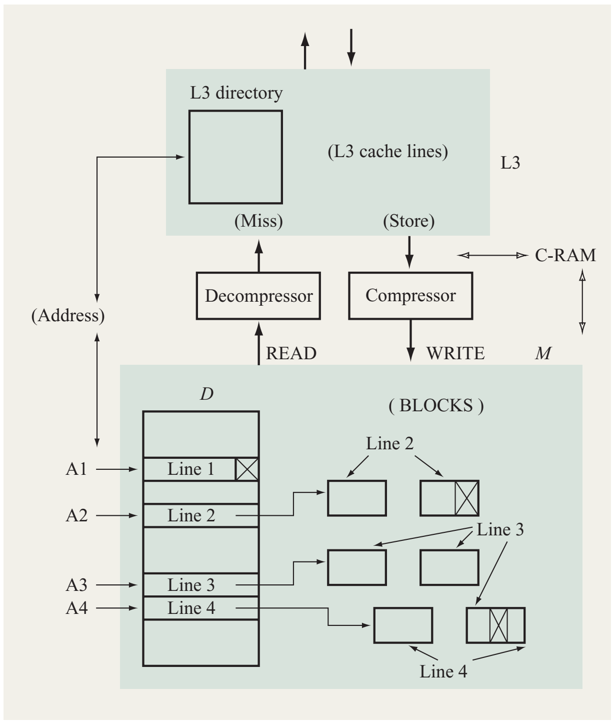

Figure 2

L3 and C-RAM organization.

by allocating the required number of blocks. As this is done by hardware, under normal operation the fact that main memory is compressed is transparent to the OS.

Given the parameters above (1KB lines and 256-byte blocks), consider the following scenario: Suppose each line compresses to 1, 2, 3, $\cdots$, or 1024 bytes, with equal likelihood. Then the expected compressed line size is 512.5 bytes; that is, compression is almost exactly 50% (50.05%). However, if some number of full blocks is used to store each line, it is easily seen that the expected number of blocks required to store a line is 2.5. This gives a compression of 62.5%, significantly worse than 50%. One way to address this problem is to make block sizes smaller. However, if block sizes are significantly smaller, the size of the directory can increase dramatically. Another approach to reducing storage fragmentation is to combine two or more *fragments*, that is, the "left-over" pieces in the last blocks used to store compressed lines, into single blocks.

In order to make fragment combining feasible using a hardware-based memory controller, and in particular to provide a guarantee of the maximum latency, some constraint must be made on the sets of lines for which fragment combining is allowed. We will call a set of lines for which fragment combining is allowed a *cohort*. In order to have a small upper bound on the time required for directory scans, ideally the size of cohorts should be small. In subsequent sections we look at cohort sizes ranging from 2 to 16 lines. Another design issue is the fashion in which cohorts are determined. Here there are two general approaches: partitioned cohorts and sliding cohorts.

In a partitioned-cohort approach, lines are divided into a number of disjoint sets (all of a given fixed size, the cohort size), where each such set is a cohort. For example, with a cohort of size 2, the first two 1KB lines in each 4KB page could form one cohort and the last two lines another cohort. Similarly, grouping lines by pages, all of the lines in each page give cohorts of size 4; all of the lines in two consecutive pages give cohorts of size 8; and so on.

In contrast, in a sliding-cohort design, cohorts are not disjoint, but overlap. For example, with a cohort of size 4, the cohort corresponding to any given line could consist of the set containing that line and the previous three lines, and similarly for other cohort sizes. The motivation for considering this type of design is the following. In a partitioned-cohort design with a cohort size of 4, as each successive line is stored sequentially, the average number of lines already stored for which combining is possible is 1.5. In a sliding-cohort design, however, again with a cohort size of 4, assuming that the lines in the previous page have already been stored, each successive line always has three other lines with which it can potentially combine. This suggests that this design might yield decreased fragmentation.

Another issue is the method by which fragments are combined. First, there is a question of the number of fragments that can be combined in a block. From a hardware point of view, the simplest case is to allow only two fragments per block. However, if significant gains would be realized by allowing three-way or greater combining, the increased complexity might be acceptable. Also, given a fragment and the sizes and locations of other fragments in the cohort for which combining is possible, which fragment (or fragments) should be chosen? Two options are to use first-fit or best-fit methods. We also study some other methods primarily for the sake of comparison, since they are impractical to implement in hardware. These are 1) fragment catenation, in which all of the fragments in a cohort are catenated, with fragments crossing block boundaries as necessary; and 2) optimal fit, in which (assuming two-way combining, for example) all possible ways of combining the fragments in a cohort are exhaustively examined, and the one yielding the minimal number of blocks is selected.

Finally, there are design issues related to choice of block size, which we consider in Section 5, and to the design of the directory structure. Here we present an overview of directory structure design; for more details, including possible directory-entry formats, addressability issues, etc., see Section 3 of [5]. There are two types of directory structures. The first is static, and is configured so as to have the required number of entries to support a maximum compression factor of F (where F is greater than 1, with $1/F$ the compression expressed as a fraction). That is, if the C-RAM has a capacity of N uncompressed lines, the directory contains entries for FN lines. Thus, for example, if F were 2 (i.e., 50% compression), D would contain entries for $2N$ lines. A possible problem with this type of design is that the maximum compression is limited to a predetermined value: If the contents of memory at some point are more compressible, this cannot be taken advantage of to provide more real memory. On the other hand, if compression is significantly less than this value, there is wasted directory space. These problems can potentially be avoided by the second type of structure we consider, in which the directory is dynamic.

Using a dynamic directory structure, directory entries are created (deleted) whenever real addresses are allocated (deallocated). In this case, free main-memory blocks could be allocated (deallocated) and used for the directory entries for one or more pages whenever the pages were created (deleted). Here, we are assuming that real memory increases (decreases) in units of at least a page. In this case, a pointer or index to the C-RAM directory entries for the page or pages would be maintained as part of the real address by the OS in a page-table entry. An interesting property of the 128-byte block case is that, assuming directory-entry formats as described in [5], one 128-byte block is exactly the right size to hold the four directory entries for the four lines of the page, provided that a maximum physical-memory addressability of 256 GB is sufficient (the maximum logical real-memory addressability would be F times as much, for example 512 GB for $F = 2$). The implication is that the C-RAM memory could be uniformly divided into a collection of 128-byte blocks, which could be allocated either as directory entries or as blocks to hold compressed data, as required.

3. Uniformly distributed compressed lines

In this and the following section we consider approaches to combining line fragments, that is, those parts of compressed lines occupying a fraction of the last block used to store the line, into single blocks. Issues include the number of fragments to be combined, the way the fragments are chosen, and whether combinations of fragments within a cohort are reorganized.

We first study these issues assuming a uniform distribution of compressed line sizes; in the next section we present results based on memory dumps. In more detail, with parameters as in previous examples (1KB line sizes, 256-byte blocks), we assume that each line is compressed to 1, 2, 3, $\cdots$, or 1024 bytes, with equal likelihood, independently of the compression of any other line. Without fragment combining, each line therefore compresses to 1, 2, 3, or 4 blocks, each with equal likelihood.

It is possible to derive some results for fragment combining for some simple cases (such as for cohorts of size 2) using analytic methods, where it is assumed that each fragment size is a real-valued random variable uniformly distributed on the unit interval, for example. However, in practice, memory at the level of the C-RAM in the memory hierarchy is accessed in certain minimum-size units, which we call *granules*. For memories of the size we are considering here, granules of size of the order of 32 bytes are appropriate. It is desirable for performance reasons to store each fragment as a number of complete granules. With 256-byte blocks and 32-byte granules, it follows that the last block used to store a line, that is, the fragment, consists of one to eight granules, with each size equally likely.

With these assumptions, it is possible to compute exactly the expected number of blocks used to combine the fragments in a cohort, given the number of granules per block, the fragment-combining method, the degree of fragment combining, and the cohort size, simply by examining all possible cases (providing the cohort size is not too large). The expected number of blocks can then be used to derive the expected compression.

In more detail, this is computed as follows. Assume that the cohort size is 4 and that there are eight granules per block, and let $F(f_1, f_2, f_3, f_4)$ be a function that computes, for a given fragment-combining method, the number of blocks required to combine four fragments of sizes f_1, f_2, f_3, and f_4 $(1 \le f_i \le 8)$. Assuming a uniform independent distribution of compressed line sizes as above, the expected number of blocks used by a compressed line *not counting the last block* is $(0 + 1 + 2 + 3)/4 = 1.5$. The expected number of blocks used for the fragments in the cohort, say X, is computed as follows:

$$X = \frac{1}{8^4} \sum_{1 \le f_1 \le 8} \sum_{1 \le f_2 \le 8} \sum_{1 \le f_3 \le 8} \sum_{1 \le f_4 \le 8} F(f_1, f_2, f_3, f_4).$$

The expected compression, computed for the entire cohort, is therefore

$$(\text{compression}) = \frac{256(1.5C + X)}{1024C},$$

where C = (cohort size) = 4. This formula is the same for any cohort size; however, for cohort sizes other than 4, the formula for computing X is changed in the obvious way.

Compression results for two-way fragment combining obtained using the above methods are shown in **Table 1**. The methods used were fragment catenation (*CAT*), first-fit (*FF2*), best-fit (*BF2*), and optimal-fit (*OF2*). As previously described, catenation and optimal-fit are shown only for comparison.

Table 1 Expected compression (%), two-way combining.

C	*CAT*	*FF2*	*BF2*	*OF2*
2	57.03	57.03	57.03	57.03
3	55.21	56.68	56.68	56.68
4	54.30	55.82	55.77	55.49
5	53.75	55.57	55.51	55.24
6	53.38	55.23	55.15	54.73
7	53.13	55.05	54.96	54.54
8	52.93	54.86	54.76	54.25

Table 2 Expected compression (%), three-way combining.

C	*CAT*	*FF3*	*BF3*	*OF3*
3	55.21	55.76	55.76	55.76
4	54.30	55.23	55.21	55.08
5	53.75	54.83	54.76	54.55
6	53.38	54.55	54.45	54.19
7	53.13	54.35	54.21	53.93
8	52.93	54.18	54.01	53.71

Exact results on expected compression, using the same set of methods, were also obtained for three-way fragment combining (up to three fragments are allowed to share a block), as shown in **Table 2** (results for the catenation method, which of course does not depend on the degree of fragment combining, are shown again for ease of comparison).

First consider the results as compared to not using combining. The expected compression using no combining is obtained by setting X to 1 above, which gives 62.5% (as in a previous example). Thus, combining gives a substantial improvement, even for the simplest case, in which the cohort is of size 2, in which the expected compression is 57.03% (for a cohort of size 2, all combining methods are equivalent, since there is no choice for selection of fragments to combine).

Next, examining various combining results, we note that first-fit, best-fit, and optimal-fit are essentially equivalent, with best-fit giving at most a fraction of a percent in compression improvement over first-fit, and optimal-fit a slightly larger but still fractional percent improvement over best-fit. Since optimal-fit is not suitable for an actual implementation (it is included only as a bound), the conclusion is that whichever method is easiest for use in a hardware-based memory controller can be used with no impact on performance. Note also that even if we remove the constraint that fragments must be combined into single blocks (i.e., the constraint that fragments not cross block boundaries), as in the catenation method, there is

Table 3 Average compressed 4KB page sizes (simulations).

Cohort size	*FF2*		*BF2*	
	Initial fill	*Steady state*	*Initial fill*	*Steady state*
4	2286 (55.8%)	2304 (56.3%)	2284 (55.8%)	2300 (56.2%)
8	2247 (54.9%)	2272 (55.5%)	2243 (54.8%)	2262 (55.2%)
16	2216 (54.1%)	2249 (54.9%)	2211 (54.0%)	2231 (54.5%)

still only a modest improvement. Using catenation, we obtain the best compression possible, subject to the constraints that the entire cohort is stored as an integral number of blocks and that each line is stored as an integral number of granules. Although this is not practical for a hardware-based memory controller, it is interesting that practical methods such as two-way first-fit or best-fit fragment combining do almost as well.

Next we examine the effects of cohort size and degree of fragment combining. Consider the results for first-fit, for example. The choice of cohort size and degree of fragment combining have an impact on the complexity and latency of operations for the memory controller. With two-way combining, compression is slightly improved (from about 55.8% to 54.9%) by increasing the cohort size from 4 to 8; that is, less than one percent improvement in compression performance is achieved. Increasing the degree of fragment combining has even less of an effect. Since we might expect this to be more significant for larger cohort sizes, consider the first-fit results for a cohort size of 8: By moving from two-way to three-way fragment combining, compression is improved only from about 54.9% to 54.2%. Overall, of the various factors related to choosing a practical design, the most improvement is seen when one moves up from a minimal cohort of size 2 to larger cohort sizes.

Simulations were also used to investigate fragment-combining methods. These were used to find average compression for larger cohort sizes (up to 16), and also validated the exact computation method results for smaller cohort sizes. Furthermore, simulations permitted studies of a C-RAM under steady-state conditions, in which after filling the C-RAM with compressed lines (with randomly generated sizes as above), lines were then selected randomly (any line equally likely), the size of the selected line changed to a new random value (again, from 1 to 1024 bytes equally likely), and then recombined (if possible) with the other lines in its cohort. In contrast, the above exact computation results correspond to what we term an *initial-fill* simulation, in which all of the lines in a C-RAM are stored in order (with multiple runs providing an average). Finally, simulations were used for preliminary studies of various alternatives in fragment recombining methods in the steady-state case, and of sliding-cohort-type designs as previously described.

Results for partitioned cohorts, where each cohort consists of one, two, or four pages, are shown in **Table 3**. These results are for two-way fragment combining, using first-fit (*FF2*) or best-fit (*BF2*) methods. Both initial-fill and steady-state results are shown.

Note that compression is slightly better in the initial-fill cases than in the corresponding steady-state cases. At first this might seem counterintuitive, since in the steady-state case the fragment of each changed line can potentially recombine with the fragment of any other line in the cohort, whereas in the initial-fill case each fragment can combine only with the preceding fragments in the cohort. However, a more detailed analysis reveals that conditions arise in the steady-state case in which fragments are not combined that would have been combined under initial fill.

Last, we comment on the general characteristics of the sliding-cohort simulation results. The results were mixed, in that slightly better compression was obtained for initial-fill cases, but slightly worse compression was obtained for steady-state cases.

4. Analysis using memory dumps

In this section we present results for three examples of real systems: main-memory dumps were obtained for 1) a 64MB AIX* system, running a memory-intensive logic simulator; 2) a 32MB Windows NT** system in which a dump was obtained immediately after the system boot process completed; and 3) a 32MB Windows NT system in which a number of different applications were started so as to allocate all of main memory. Below, these are referred to respectively as the AIX(active), NT(boot), and NT(active) memory dumps. A program emulating a four-way parallel shared-dictionary compression method [7] was used to compress each memory dump, one line

(1024 bytes) at a time, resulting in a sequence of compressed- line sizes. Each sequence of compressed-line sizes was then used as input to various “initial-fill”-type simulations, as described in the previous section.

Results for the AIX and NT dumps are shown in **Tables 4**, **5**, and **6** for block sizes of 64, 128, and 256 bytes, respectively; in all cases the granule size is 32 bytes. A block size of 64 bytes is a special case: With 32-byte granules, each fragment is either full or half full. Given a cohort, the same number of blocks are produced by any of the fragment-combining methods in this case (including catenation): If there are an even number of half-full fragments in a cohort, they all combine pairwise; otherwise, with an odd number, all but one fragment combine pairwise, leaving exactly one half-full fragment. Therefore, in Table 4 we give (for each memory dump and cohort size) one compression result. For the larger block sizes we give compression results for FF2 and FF3; best-fit results were also obtained but are not shown because, to the accuracy used in the tables (0.1%), the compression results for best-fit are essentially the same as for first-fit.

As discussed in the previous section, the compression obtained using catenation is the best possible, subject to the constraints that each line is stored as an integral number of granules and that each cohort is stored as an integral number of blocks. In fact, the compression results shown in Table 4 (for a block size of 64 bytes) are quite close to the “raw” line size compressions; that is, they are close to the average of the compressed-line sizes used as input to the initial-fill-type simulation divided by 1024. For comparison, these are 34.1% for the AIX memory dump, 48.8% for the NT(boot) dump, and 36.3% for the NT(active) dump. However, even though the compression results shown here are best for the 64-byte block size, we see in the next section that this is not the best choice of block size when the required directory space is taken into account.

In contrast to the results based on a uniform distribution of compressed-line sizes, note that there is a significant improvement from using three-way combining for a block size of 256 bytes (the improvement is less for the smaller block sizes). An analysis of the distribution of line sizes for the various dumps does in fact show a strong degree of nonuniformity, with sharp peaks for a number of small compressed-line sizes. For example, in the case of the AIX dump, out of 65 536 lines, the most common compressed-line size is 13 bytes (corresponding to a line consisting of the same repeated byte, e.g., a line consisting of all nulls); such lines occur 5828 times; i.e., they comprise approximately 9% of all lines. Owing to the presence of these and other small compressed-line sizes, one would expect an advantage in being able to combine more than two fragments. However, as noted previously,

Table 4 Compression (%) of memory dumps, block size = 64.

Cohort size	*AIX(active) CAT, FF, BF*	*NT(boot) CAT, FF, BF*	*NT(active) CAT, FF, BF*
4	36.0	50.7	38.2
8	35.8	50.5	38.1
16	35.7	50.4	38.0

in a fixed-size directory entry design, it is possible to store sufficiently small compressed lines entirely in the directory entry for the line. With 256-byte blocks, directory entries could consist of four pointers plus flag bits, etc., and require 16 bytes; in such a case, lines compressing to 15 bytes or less could be stored entirely within the directory entry. Thus, in practice, three-way combining might not be as effective as indicated above, since the small compressed lines that give the improvement would not be available for combining when stored within directory entries. In order to investigate this, results were obtained for the 256-byte block case with the following modification: The compressed-line size for all lines compressing to 15 bytes or less was reset to 0 (emulating storing the line entirely within the directory entry). The results are shown in **Table 7**.

5. Optimal block sizes

In this section we consider the problem of choosing an optimal block size. The tradeoff involved in choosing a block size is that the larger the block size, the more wasted space there is due to fragmentation; the smaller the block size, the more space is required for the directory.

In order to obtain an analytic result, we make some simplifying assumptions. One of these is that the C-RAM has a fixed average compression (that is, we assume that the average compression is a constant). In the case of a static-directory structure, our analysis applies to the case in which the average compression is set to the maximum level of compression supported by the directory structure; the resulting block size will be (subject to the simplifying assumptions) that which minimizes the total required space for a static-directory design with the given maximum compression parameter. In the case of a dynamic-directory structure, the average compression should be set to a value which is expected to be close to that which will occur in practice; the resulting block size will be approximately optimal (in terms of total required space) when the C-RAM is operating at this degree of compression. The following analysis actually applies to any storage structure in which a number of fixed-size blocks are used to store objects of variable size; therefore, we use the term “object” to refer to a variable-size entity that is

Table 5 Compression (%) of memory dumps, block size = 128.

Cohort size	*AIX(active)*		*NT(boot)*		*NT(active)*	
	FF2	*FF3*	*FF2*	*FF3*	*FF2*	*FF3*
4	38.1	37.2	53.4	52.0	41.2	39.5
8	37.8	36.7	53.0	51.3	40.8	38.8
16	37.5	36.4	52.7	51.0	40.5	38.5

Table 6 Compression (%) of memory dumps, block size = 256.

Cohort size	*AIX(active)*		*NT(boot)*		*NT(active)*	
	FF2	*FF3*	*FF2*	*FF3*	*FF2*	*FF3*
4	43.3	40.4	59.0	55.0	47.4	42.6
8	42.7	38.9	58.3	53.3	46.7	40.8
16	42.1	38.3	57.7	52.5	46.1	40.0

Table 7 Compression (%) of memory dumps, block size = 256 (lines compressing to 15 bytes or less in directory).

Cohort size	*AIX(active)*		*NT(boot)*		*NT(active)*	
	FF2	*FF3*	*FF2*	*FF3*	*FF2*	*FF3*
4	41.1	39.5	55.2	53.5	43.3	41.1
8	40.5	38.3	54.5	52.3	42.6	39.8
16	39.9	37.6	54.0	51.7	42.0	39.1

being stored; in the context of C-RAM design, each object is a compressed line.

Given a memory of total size M bytes, this memory contains a directory which, for each object (i.e., compressed line in the C-RAM context), gives the location of the blocks used to store the object. Here we use the term “directory” in a general fashion: The directory consists of all data in the memory other than blocks. For example, a simplistic “directory structure” is the following: 1) external tables provide a pointer to the first block used to store a given object; 2) each block is extended with a pointer field, which is used to point to the next block used to store the object, or is null if the block is the last such block used. In this example, the collection of all pointers is considered to be the directory.

Let q be the number of bytes used by the directory (in the above general sense) per block stored in the memory. For the previous simple design example, determining q is straightforward: q is just the length (in bytes) of a block pointer. In other cases, estimating q can require additional information. For example, in the case of a directory structure as previously illustrated in Figure 1, each directory entry could have four pointers to blocks, plus flags and fragment-combining information. Suppose this directory entry requires q' bytes. In order to estimate q, we also need to know the average compression. If the average compression is 50%, say, then even though the average number of blocks pointed to by directory entries could be greater than 2 (due to fragment combining), the total number of blocks is half the total number of pointers in the directory. In this case, q would be $q'/2$. If, for example, each directory entry were 16 bytes long, q would be 8 bytes; if, on the other hand, compression were 25%, q would be 16 bytes, and so on. Although formally q could be considered to be a function of block size and possibly other parameters, the above examples illustrate that in practice q can be estimated easily given a particular directory structure, together with (in some cases) the assumed constant average compression. Therefore, we assume that q is a constant; the implications of this simplifying assumption are discussed below.

Next, let p be the average size (in bytes) of an object, and let b be the block size. Our objective is to find a value of b that maximizes the total number of objects that can be stored in a memory of size M (equivalently, as shown below, given the total number of objects, we will find a value of b that minimizes the memory size M required to store the objects). In order to do this, we need to consider the effect of fragment combining. Suppose there were no fragment combining. Then, assuming a continuous uniform distribution of fragments, the expected number of blocks to store an object would not be simply p/b, but rather $p/b + 1/2$, since the last block used to store an object (under these assumptions) is on the average half full. If pairs of fragments were catenated, then again, assuming a uniform continuous distribution of fragment sizes, it can be shown that the expected size of a fragment would be 1/2 of a block. However, now this fragment would be shared between two objects, so in this case the expected number of blocks to store a compressed object would be $p/b + 1/4$. We parameterize the effect of fragment combining as follows: Given a particular fragment-combining scheme, if the average number of blocks to store an object under this scheme is found to be $p/b + 1/n$, we call n the *fragment-combining effectiveness*. For fragment-combining methods described in previous sections, such as two-way combining using first-fit or best-fit with a cohort size of 4 and 32-byte granules, n can be computed and has been found to be approximately 4 (e.g., n is found to be 4.3 for FF2 using the values from Table 1 of Section 3).

Given the total memory size M, as discussed earlier, the tradeoff in finding an optimal value for the block size b is as follows: As b increases, there is more wasted space in fragments, since on a per-object basis each fragment is on the average a fraction $1/n$ of a full block; as b decreases, there is a larger total number of blocks, and therefore more overhead in directory space, since each block requires q bytes of directory space. An estimate of the optimal value of b can be found as follows, using a continuous approximation. First, the total number of blocks in the memory is

$$\frac{M}{b+q}.$$

Next, the total number of objects T that can be stored in the memory as a function of b is given by the following:

$$T(b) = \frac{M/(b+q)}{p/b + 1/n} = \frac{Mnb}{(b+q)(np+b)}.$$

From this expression, it is easily found (see [4]) that, under the above assumptions, the optimal value of b is given by the following:

$$b_{\text{opt}} = \sqrt{npq}.$$

As an example, suppose we use a directory structure for a C-RAM with directory entries consisting of K block pointers (which includes flags and fragment-combining information), and suppose each such entry requires $4K$ bytes (where K depends on the block size; that is, smaller blocks require more pointers for a given fixed line size in the case that the line cannot be compressed). As in the discussion above, under 50% compression this gives a value for q of approximately 8 bytes (note that this is independent of K). Setting p to 512 bytes (for 50% compression of 1024-byte lines), and setting n to 4 (which is typical for two-way combining using first-fit or best-fit with granules of size 32 bytes), we find

$$b_{\text{opt}} \approx \sqrt{4 \times 512 \times 8} = 128.$$

Note that the equation above can be rewritten as $M/T = (b + q)(np + b)/nb$. M/T represents the total bytes used (including directory space) per object, and b_{opt} minimizes this quantity. This means that if instead of starting with a fixed memory size M, we start with a fixed number of objects T (corresponding to a given number of lines, i.e., a logical memory size in the C-RAM context), b_{opt} minimizes the total memory required to store the given number of objects.

For C-RAM directory structures of the types previously discussed, in which highly compressible lines can be stored entirely within directory entries, in practice there is a complicating factor: As the block size decreases, the size of directory entries increases, and therefore the maximum compressed-line size that can be stored in a directory entry also increases. Since such lines require no blocks to be allocated, one might expect the improvement as the block size is decreased to the optimal size as determined from the above to be greater than that predicted by analysis; similarly, one might expect the improvement as the block size is increased to the optimal size as given above to be less than that predicted analytically.

In order to investigate this issue, compression using the memory dumps described in the previous section was found for block sizes of 64, 128, and 256 bytes, in which, however, lines compressing to 63, 31, or 15 bytes, respectively, were considered to be stored entirely in directory entries (that is, the compressed size of such lines was set to 0). For the NT(active) case, for example, with a cohort size of 4, and using FF2 or BF2, and with other parameters such as granule size set as previously described, compressions of 37.3%, 38.6%, and 43.3% were obtained for the 64-, 128-, and 256-byte block-size cases, respectively. For the AIX(active) case, the corresponding compressions obtained were 35.1%, 36.6%, and 41.1%; for the NT(boot) case, the corresponding compressions were 50.0%, 51.1%, and 55.2%. Suppose that we are using a dynamic-directory structure and wish to provide a given logical memory size: How much total memory (including the directory) would be required for each of these cases?

Table 8 Required total memory for each megabyte of logical memory.

Block size	*AIX(active)* (KB)	*NT(boot)* (KB)	*NT(active)* (KB)
256	436.9	581.2	459.4
128	406.8	555.3	427.3
64	423.4	576.0	446.0

Assuming that directory entries are 64, 32, and 16 bytes per line for block sizes of 64, 128, and 256 bytes, respectively, the total required memory for the various compressions given above can be computed as follows. Each megabyte of logical memory consists of 1024 1KB lines, which requires 64 KB, 32 KB, and 16 KB of directory space for block sizes of 64, 128, and 256 bytes, respectively. The space used by blocks to store compressed lines, for each megabyte of logical memory, is 1 MB multiplied by the compression number given above. The results are as shown in **Table 8**.

Here, a block size of 128 bytes (as estimated analytically above) is optimal among the cases in which block sizes are constrained to be powers of 2. As one moves from a block size of 256 bytes to 128 bytes, approximately 7% less memory is required [the improvement is slightly less for the NT(boot) case]; moving from 64-byte blocks to 128-byte blocks, approximately 4% less memory is required.

Previously we defined compression as the sum of the sizes of the blocks used to store a set of lines divided by the sum of the uncompressed line sizes. If, however, we include the space used by the directory, which is necessary to address the lines, we obtain a larger number, which we term the *net compression*. To distinguish this from our earlier definition, we call the compression obtained not including the directory space the *data compression*. It is of interest to see how much compression is lost due to the inclusion of the space used by the directory. A comparison of the data and net compressions for the cases shown previously in Table 8 is shown in **Table 9**, in which the net compressions are found from the values given in Table 8 by dividing by 1024 KB. With a block size of 256 bytes, the loss in compression due to including directory space is quite low; at a block size of 128 bytes there is a greater corresponding loss, but due to improved data compression, the net compression is better than the 256-byte block size case; finally, with 64-byte blocks, the loss is substantial, resulting in worse net compression as compared to the 128-byte block case even with somewhat better data compression.

Similar results were obtained for static-directory designs. Analysis of various static-directory-design cases (not shown here) indicates improvements from using 128-byte blocks versus 64- or 256-byte blocks in total required memory ranging from 4% to 13%, where the maximum compression parameter is set to net compression values ranging from 25% (4:1) to 50% (2:1).

6. Conclusions

Effective use of compressed random-access memories at the lowest level of a system main-memory hierarchy involves finding relatively simple designs, suitable for hardware implementation, for directory structures and memory management, which nevertheless make efficient use of available memory. With respect to memory management, since cache lines compress to varying sizes, an immediate problem is that of allocating storage for variable-size objects. The simple approach of organizing the memory as a collection of fixed-size blocks, and allocating an integral number of blocks to store each compressed line, results in significant fragmentation for practical block sizes. However, by allowing the last partially used blocks, i.e., fragments, allocated to small predetermined sets of lines, called cohorts, to be combined, fragmentation can be significantly reduced. Neither large cohort sizes nor high degrees of fragment combining are necessary; the results presented here indicate that very little benefit is obtained beyond using two-way combining with cohorts of size 4, for example.

With respect to directory structure, there are two approaches: In a static design, the directory is preallocated with a fixed size sufficient to support a given maximum degree of compression; in a dynamic design, groups of directory entries (for the lines contained in a page, for example) are allocated and deallocated as compression, and therefore the total logical main-memory size, varies. The advantage of the static design is simplicity; the disadvantage is that unless compression is close to the

Table 9 Comparison of data compression and net compression (%).

Block size	*AIX(active)*		*NT(boot)*		*NT(active)*	
	Data	*Net*	*Data*	*Net*	*Data*	*Net*
256	41.1	42.7	55.2	56.8	43.3	44.9
128	36.6	39.7	51.1	54.2	38.6	41.7
64	35.1	41.3	50.0	56.3	37.3	43.6

predetermined maximum value, there is wasted space (if compression is better than the predetermined maximum, there are unused blocks; if compression is worse than this value, there are unused directory entries). An interesting property of the dynamic-directory design is that, assuming 4KB pages and 1KB lines, a block of size 128 bytes can be used as either a group of four directory entries for a page or as a block to hold compressed-line data, and that, as described below, this block size is optimal under certain general assumptions. Thus, in this case memory could be managed in a uniform way as a collection of 128-byte blocks, used for either directory entries or data as required (as opposed to approaches in which memory is partitioned into directory and data spaces).

With respect to choice of block size, the tradeoff is that smaller block sizes require a larger directory space; however, larger blocks result in increased fragmentation. Thus, finding an optimal block size involves balancing the required directory space and amount of fragmentation. An analytic model was used to predict the optimal block size, given the average compressed-line size, the bytes of directory space used per block, and a measure of fragment-combining effectiveness. For typical parameters and fragment-combining effectiveness results, we found that the optimal block size is of the order of 128 bytes. However, the analytic model uses several simplifying assumptions, and in particular does not take into account the fact that larger directory entries can be used to store more lines in a format in which no blocks are allocated for a line when the compressed line can fit entirely within the directory entry. Interestingly, a more detailed analysis (that included this effect) using memory dumps to compare various block sizes yielded the same result as the analytic model; i.e., a block size of 128 bytes was found to be optimal.

Finally, although the dynamic-directory approach is somewhat more complex than the static design, it appears that it is also suitable for hardware implementation. However, if, in a given system, the compression does not vary greatly, a static design (configured appropriately) could be acceptable.

Acknowledgment

Charles Schulz and Dan Poff gave significant help in providing parallel compression emulation code and access to memory dumps.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Microsoft Corporation.

References

1. M. Pietrek and L. Seltzer, "Windows 3.1 Memory Enhancement Utilities," *PC Magazine* **15,** No. 4, 205–213 (February 20, 1996).
2. M. Kjelso, M. Gooch, and S. Jones, "Design and Performance of a Main Memory Hardware Data Compressor," *Proceedings of the 22nd EUROMICRO Conference*, IEEE, 1996, pp. 423–430.
3. 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.
4. 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).
5. P. Franaszek and J. Robinson, "Design and Analysis of Internal Organizations for Compressed Random Access Memories," *Research Report RC-21146*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, October 20, 1998.
6. 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.
7. P. A. Franaszek, "A System and Method of Compression and Decompression Using Store Addressing," U.S. Patent 5,864,859, January 26, 1999.
8. 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.
9. 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.

Received August 29, 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 IBM 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.

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.