RC 21146(94535)20OCT98 Computer Science/Mathematics

Research Report

Design and Analysis of Internal Organizations for Compressed Random Access Memories

Peter A. Franaszek, John T. Robinson
IBM Research Division
T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
{paf, robnson}@watson.ibm.com

Design and Analysis of Internal Organizations for Compressed Random Access Memories

Peter A. Franaszek and John T. Robinson
IBM Research Division, T. J. Watson Research Center
P. O. Box 218, Yorktown Heights, NY 10598
{*paf, robnson*}*@watson.ibm.com*

March 30, 1998
(revised October 9, 1998)

Abstract

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 requirements 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, presented 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 evaluate various approaches using analytic methods and simulations.

1. Introduction

With higher processor speeds, and a lack of a corresponding speedup in disk access, the tendency in server-class computers is towards increasingly large main memory sizes in proportion to processor speeds. 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 costs. A possible approach to mitigating this cost is to have the contents of main memory be compressed. It is well known that such contents are generally compressible by a factor of two 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 where 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 re-compressed on write-backs), it is necessary to develop some new technology in order to obtain a practical system.

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 LZ methods. In particular, parallel, shared dictionary techniques [4], 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 [5]. Last, a way must be found for efficiently storing and accessing 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 is comprised of 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 (this is only approximately true, since as we describe later highly compressible lines may be stored entirely within directory entries). 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 the lowest level (i.e. M, or L4 in the current context) of the memory hierarchy. In the following, we 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 [4], and estimated L3 cache miss latencies. The larger the L3 cache line, the greater the latency, but the better the compression.

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/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. Methods of the first two types can be attractive for some applications (e.g. allocation of extents in disk storage, and the class of main memory allocation techniques referred to as buddy systems). However, each has the property of eventually requiring reorganization or garbage collection due to space fragmentation or unfavorable distributions of block sizes. These processes can be time consuming and complex, and are not well-suited to applications where fast response times and simple processing are required.

The techniques we investigate here are of the third type. We use blocks of a single fixed size, but permit sharing of blocks between variable length objects. The methods actually considered are a simple form of this notion. The blocks are of uniform 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 cohorts and 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.

The following is a synopsis of the paper. In Section 2 we present a high-level design overview, and discuss various design issues. A difference between a standard design and one with compressed memory is that the "real addresses" are used as pointers or indices to directory entries describing the compressed format for each line. Section 3 describes two approaches to the design of such directories, which can respectively be characterized as static and dynamic. Section 4 considers a variety of techniques for block sharing under the simplifying assumption that the compressed line sizes are uniformly distributed. The results suggest that satisfactory performance can be obtained by limiting the number of lines sharing a block to two, with cohorts of size four. Section 5 shows some experimental results for main memory dump data drawn from various systems. Section 6 considers the question of finding an optimal fixed block size from the point of view of maximizing the number of variable size objects that can be stored. This involves a tradeoff between fragmentation and directory size. The optimal block size is found to be on the order of 128 bytes for systems similar to those described in earlier sections. The overall result is that storage losses due to factors such as fragmentation and inclusion of a directory structure increase the amount of storage required by roughly 15% over the raw compression amount, for example yielding a net compression of 41.7% versus an initial compression of 36.3% for a Windows NT memory dump. Conclusions and some discussion are provided in Section 7.

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 1. 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 1. The C-RAM memory consists of two parts, the directory D and a collection of fixed sized blocks. Highly compressible lines may be stored entirely within directory entries. Otherwise, the directory entry points to one or more of the fixed sized 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. An important feature is that information pertaining to the blocks holding a compressed L3 line (or the compressed line itself in the case that the line fits entirely within a directory entry) is stored at a location which can be determined, e.g. using a known offset, from the real address.

As an example, suppose that pages are of size 4K bytes, and that the L3 cache immediately above the C-RAM has a line size of 1K bytes. Unlike 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 are of size 256 bytes (the issue of choosing an optimal block size is discussed in Section 6). Each directory entry contains flags, fragment combining information, and assuming a fixed sized entry directory structure (as explained in Section 3), 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), then it is stored in an uncompressed format, as indicated by a flag in the directory entry. Finally, as previously discussed, 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 decompresses 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), the line is compressed, and then stored in the C-RAM 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. However, since memory is compressed, the logical total size of main memory can grow or shrink dynamically, and the OS must be modified to handle this. This latter issue

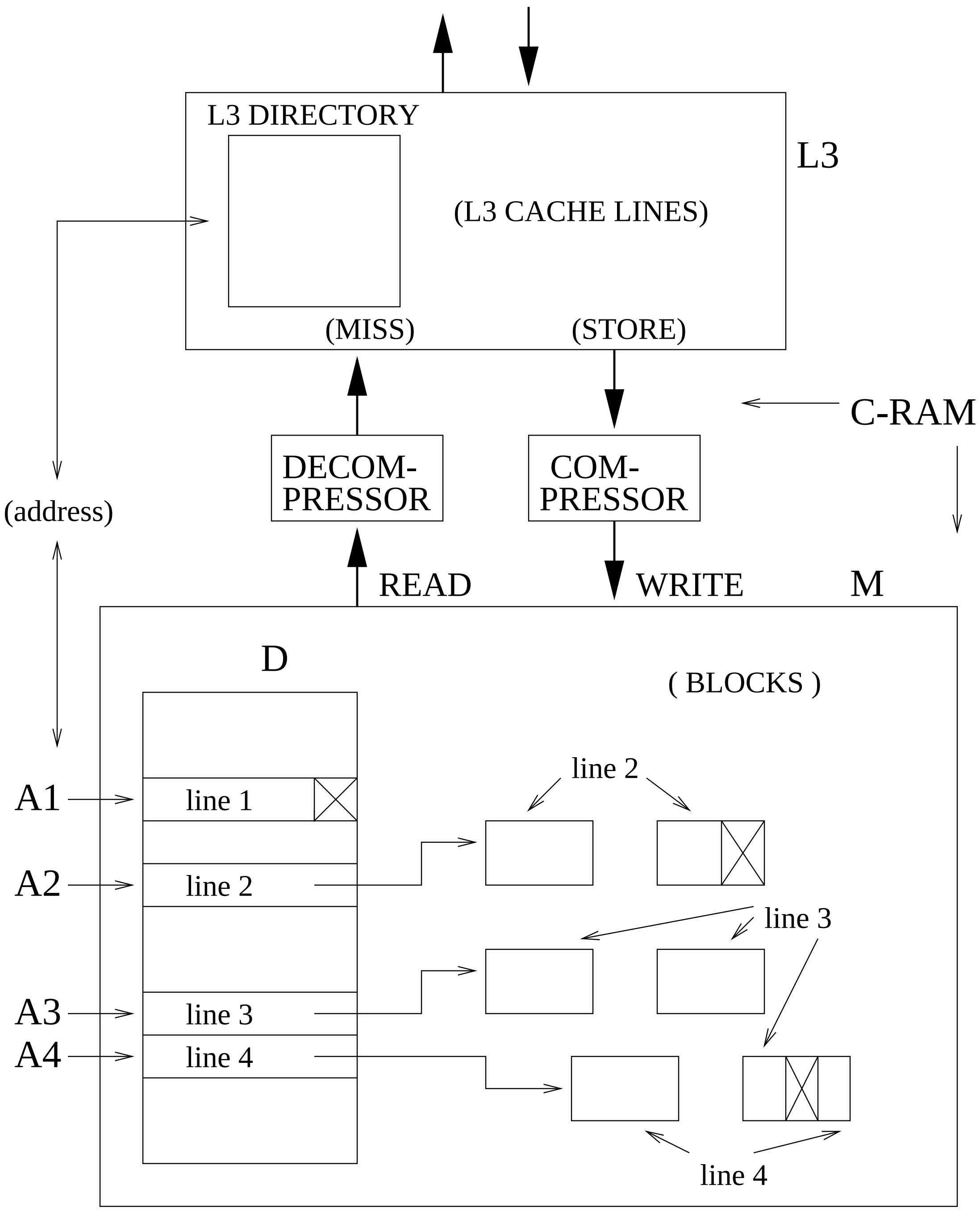

Figure 1. System Overview

is outside of the scope of the current paper, and is addressed in [5]; here we will focus on the internal structure and performance of the C-RAM.

Given the parameters above (1K byte lines and 256 byte blocks), consider the following scenario: suppose each line compresses to 1,2,3, ..., or 1024 bytes, equally likely. Then the expected compressed line size is 512.5 bytes, that is compression is almost exactly 50% (50.05%). However, using some number of full blocks to store each line, it is easily seen that 1, 2, 3, or 4 blocks are equally likely, that is, the expected number of blocks 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 (as discussed in Section 6). 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 section 6 we return to the problem of choosing block sizes, however our analysis there includes a measure of the effectiveness of fragment combining).

In order to make fragment combining feasible using a hardware-based memory controller, and in particular to provide a guarantee on 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 will 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 1K byte lines in each 4K byte page could form one cohort and the last two lines another cohort. Similarly, grouping lines by pages, all the lines in each page give cohorts of size 4; all the lines in two consecutive pages give cohorts of size 8; and so on.

In contrast, in a sliding cohort type of 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 3 lines, and similarly for other cohort sizes. The motivation for considering this type of design is the following. Consider a partitioned cohort design, with the 4 lines in each page making up a cohort, and suppose the 4 lines in a page are new lines being stored sequentially in the C-RAM. Then the first line has no other lines with which to combine; the next line has one other line with which to potentially combine; and so on; the average number of already stored lines that can potentially be combined with each of the four lines is $(0+1+2+3)/4 = 1.5$. In a sliding cohort design, however, again with a cohort size of 4, then assuming the lines in the previous page have already been stored, each successive line always has 3 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 how many fragments 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 3-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 the fragments in a cohort are catenated, with fragments crossing block boundaries as necessary; and (2) optimal fit, in which (assuming 2-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 6, and to the design of the directory structure, which we consider in the next section. One approach is to allocate a predetermined amount of memory in M to be used for the directory D. A possible problem with this type of design is that there may be significant wasted space. An alternative approach is a dynamic structure, as we subsequently discuss.

3. Directory Structure

We consider 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 one, 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 pre-determined 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).

Each directory entry contains pointers to blocks holding the compressed data. Also in the directory entry may be some subset of the compressed data, as well as various flags. The requirement that the location of the entry describing the storage of a line be obtainable by computation from a real address requires that each entry in the directory be of a fixed size (independent of the compression of each corresponding line), as otherwise the line entry would not be at a fixed displacement from the directory entry pointer for the page.

Figure 2 illustrates some possible formats for the descriptors of four 1K byte lines making up a 4K byte page in a directory. For the present we will assume a block size of 256 bytes. In this example, the descriptor for each line has one byte of flag data, and enough space to hold the addresses (3.75 bytes or 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). 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, 2 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 are assuming 2-way combining), and 3 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 2 has a format which includes the addresses for all the blocks holding a line. In contrast, line 2 shows an alternate format where 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. In this type of design, each block would contain compressed data followed by a pointer to the next

<table>
<tr><td>LINE 1</td><td>FLAGS</td><td>BLOCK 1
ADDRESS</td><td>BLOCK 2
ADDRESS</td><td>BLOCK 3
ADDRESS</td><td>BLOCK 4
ADDRESS</td></tr>
<tr><td>LINE 2</td><td>FLAGS</td><td>BLOCK 1
ADDRESS</td><td colspan="2">BEGINNING OF
COMPRESSED LINE</td><td>FRAG.BLK.
ADDRESS</td></tr>
<tr><td>LINE 3</td><td>FLAGS</td><td colspan="4">COMPRESSED LINE (SIZE <= 120 BITS)</td></tr>
<tr><td>LINE 4</td><td>FLAGS</td><td colspan="4">. . .</td></tr>
<tr><td>BITS</td><td>8</td><td>30</td><td>30</td><td>30</td><td>30</td></tr>
<tr><td></td><td colspan="5">16 BYTES</td></tr>
</table>

FLAGS

(A) COMPRESSED/UNCOMPRESSED (1 BIT)
(B) STORED IN DIRECTORY (1 BIT)
(C) NUMBER OF BLOCKS USED (2 BITS)
(D) FRAGMENT INFORMATION (4 BITS)

Figure 2. Directory Entry Formats

block, if any. The second block address in the directory entry for line 2 is for that of the block holding the fragment. This may be required so that the memory controller need not follow pointer chains in order to find the necessary blocks for fragment combining, which could cause significant delays. Finally, the entry for line 3 in Figure 2 illustrates the case in which the compressed line can be stored entirely within the directory entry.

The format given by line 2 in Figure 2 saves space associated with block addresses when these are not required. If the compressed line can fit entirely into the space allocated to the directory entry, there is no need for a pointer to following blocks (as illustrated by line 3 in Figure 2), thus the block 1 and fragment block addresses need not be included (this also applies to the format given by line 1 in Figure 2). Having at most the first block and fragment block addresses in the directory entry saves space associated with block pointers when these are not needed. This becomes more important if the block size is decreased (and the uncompressed line size remains fixed), so that on average there are more blocks. However, this format may require somewhat greater hardware complexity. Accordingly, we assume in the following that the formats given by lines 1 and 3 of Figure 2 will be used.

Suppose that, as in Figure 2, the directory entry for a 4K byte page with four 1K byte lines requires 64 bytes, i.e. 16 bytes per line. With one byte used for flag data, this leaves 30 bits (as shown) for each block address, yielding 256 GB of addressability with 256 byte blocks. If the block size is reduced to 128 or 64 bytes, the space required for the flag data would remain at one byte per line: field C of Figure 2 increases but field D decreases by the same amount. Suppose the directory entry size is increased by corresponding amounts for 128 or 64 byte block sizes, that is to 32 or 64 bytes per line, respectively. This would yield $31 \times 8/8 = 31$ and $63 \times 8/16 = 31.5$ bits for pointers for each block size. However, since we can't use a fractional bit for all of the 16 pointers that may be required for the 64 byte block case, in practice this means we would have 31 bits for pointers for both cases. This yields 256 GB of addressability for the case of 128 byte blocks (i.e. the same as that using 256 byte blocks), but 128 GB of addressability for the case of 64 byte blocks.

As described above, the directory can be either static or dynamic. In the static case, a fixed portion of memory would be allocated to hold the directory. In the dynamic case, free blocks could be allocated (de-allocated), 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, as mentioned earlier, 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 4K byte pages and 1K byte lines as above, 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 addressability of 256 GB is sufficient. 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 blocks to hold compressed data, as required. In view of the previously discussed advantages of the dynamic directory approach (no necessity to impose a limit on the maximum compression, and no wasted space associated with unused directory entries), together with optimality results for 128 byte blocks (see Section 6) and the fact that there is no loss of addressability (as compared to 256 byte blocks), this appears to be an attractive design alternative, providing the additional complexity inherent in dynamic directory entry allocation is acceptable.

4. 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 (1K byte line sizes, 256 byte blocks), we assume that each line is compressed to 1, 2, 3, ..., or 1024 bytes, equally likely, independently of the compression of any other line. Without fragment combining, each line therefore compresses to 1, 2, 3, or 4 blocks, each equally likely.

It is possible to derive some results for fragment combining for some simple cases (such as for cohorts of size of 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 level of the C-RAM in the memory hierarchy is accessed in certain minimum sized units, which we call *granules*. For memories of the size we are considering here, granules of size on 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 from 1 to 8 granules, 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 granules per block, fragment combining method, degree of fragment combining, and cohort size, simply by examining all possible cases (providing the cohort size is not too large). For example, with cohorts of size 4, and with 8 granules/block, there are $8^4 = 4096$ cases. With cohorts of size 8 (the largest cohort size for which we present exact results), there are $8^8 = 16777216$ cases. Note that in general all cases need be examined, since for fragment combining methods such as first-fit or best-fit the ordering is significant (for optimal fit, the computational complexity could be reduced using symmetries; for fragment catenation, the expected number of blocks can actually be directly computed). 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, that there are 8 granules/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 $f_1, f_2, f_3, \text{and} f_4$ $(1 \leq f_i \leq 8)$. 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.

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

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.

Results for two-way fragment combining are shown in Tables 1 and 2. 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. In Table 1 the expected number of blocks used by the fragments in a cohort after fragment combining are shown (i.e. the value X above); in Table 2 the resulting compression is shown.

C	CAT	FF2	BF2	OF2
2	1.562	1.562	1.562	1.562
3	2.125	2.301	2.301	2.301
4	2.688	2.931	2.923	2.878
5	3.250	3.613	3.602	3.548
6	3.812	4.254	4.235	4.134
7	4.375	4.914	4.889	4.771
8	4.938	5.554	5.522	5.361

Table 1. Expected Blocks for Combining Fragments (2-Way)

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 (%), 2-Way Combining

Exact results on expected compression, using the same set of methods, were also obtained for 3-way fragment combining (up to three fragments are allowed to share a block), as shown in Table 3 (results for the catenation method, which of course does not depend on the fragment combining degree, are shown again for ease of comparison).

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

Table 3. Expected Compression (%), 3-Way Combining

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 two, in which the expected compression is 57.03% (for a cohort of size two, 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 without a performance impact. 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 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 2-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 2-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 2-way to 3-way fragment combining, compression is only improved 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.

Exact results using the model above were also computed for the case corresponding to a block size of 128 bytes, with all other parameters as before. In this case there are only four granules per block, which gives slightly larger numbers for the expected number of blocks required to combine fragments as compared to those of Table 1 (which corresponds to the 256 byte block case). However, since each line now compresses to a number of blocks from 0 to 7 (not counting the last block), equally likely, the formula for computing compression becomes

$$(\text{compression}) = \frac{128(3.5C + X)}{1024C},$$

where C is the cohort size and X is as before. The result is that even with slightly larger values for X, there is still an overall improvement in compression. For example, using first-fit with 2-way combining and a cohort of size 4, compression improves from 55.8% with a block size of 256 to 53.3% with a block size of 128. One tradeoff is that with a fixed size directory entry design as in Figure 1, the directory is now twice as large; we will return to this issue in Section 6, in which this tradeoff is studied explicitly using an analytic model together with results based on memory dumps. The key point with respect to the validity of the analytic model is that the fragment combining effectiveness (as defined in Section 6) does not seem to change significantly as the granules per block is varied among values that might occur in practice.

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), its size changed to a new random value (again, from 1 to 1024 bytes equally likely), and then re-combined (if possible) with the other lines in its cohort. In contrast, the above exact compuation results correspond to what we term an *initial fill* simulation, in which all 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 re-combining methods in the steady state case, and of sliding cohort type designs as previously described.

Results for partitioned cohorts, where each cohort consists of 1, 2, or 4 pages, are shown in Table 4. These results are for 2-way fragment combining, using first-fit ("FF2") or best-fit ("BF2") methods. Both initial fill and steady state results are shown.

	FF2		BF2	
COHORT SIZE	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%)

Table 4. Average Compressed 4K Page Sizes (Simulations)

Note that compression is slightly better in the initial fill cases as compared to the corresponding steady state cases. At first this might seem counter-intuitive, since in the steady state case the fragment of each changed line can potentially re-combine with the fragment of any other line in the cohort, whereas in the initial fill case each fragment can only combine with the preceding fragments in the cohort. However a more detailed analysis reveals that cases arise in the steady state case in which fragments are not combined that would have been combined under initial fill. One reason for this is due to fragments being "orphaned". For example, suppose the fragment of line 1 is combined with the fragment of line 2 in the steady state case, and that line 2 is subsequently changed so that its fragment size increases to a point where it cannot be combined with any other fragment. Then without further work, the fragment of line 1, which could be small and readily combined with the fragment from a line other than line 2, is left uncombined (i.e. orphaned). These considerations lead to various additional methods for fragment re-combining. The simplest version, which we call partial re-combining, is as follows: when a line is changed, if its fragment were previously combined with the fragment of a line L, and after the change its fragment cannot be combined with the fragment of L, then fragment re-combining is applied to both the changed line and to L. Simulation results indicate that partial re-combining yields compression close to initial fill (given corresponding parameters and combining methods). "Full" re-combining, in which each changed line would cause all fragments in the cohort to be re-combined from scratch, would of course be equivalent to an initial fill simulation. However, using full re-combining, each change to a line would incur significant processing, and therefore might be impractical due to hardware complexity.

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 slighly worse compression was obtained for steady state cases (as compared to the partitioned cohort results, with corresponding parameters and combining methods). For example, for a cohort size of four, compressions of 54.3% and 56.9% were obtained using sliding for the intial fill and steady state cases, respectively (using either FF2 or BF2).

These compare with the Table 4 results of 55.8% and 56.3% (for FF2; 56.2% for BF2) for the same cases using partitioned cohorts. Thus, the compression results were generally close. The overall conclusion from the analysis and simulations using a uniform distribution of compressed line sizes is that given a reasonable cohort size (of four, say), other design considerations, such as the hardware complexity associated with each method or parameter choice, would likely be of greater importance than fairly small variations in compression.

5. Analysis using Memory Dumps

In this section we present results for three examples of real systems: main memory dumps were obtained for (1) a 64 MB AIX system, running a memory-intensive logic simulator; (2) a 32 MB NT system, in which a dump was obtained immediately after the system boot process completed; and (3) a 32 MB NT system in which a number of different applications were started so as to allocate all of main memory. Below, these are referred to as the AIX (active), NT (boot), and NT (active) memory dumps, respectively. A program emulating a 4-way parallel shared dictionary compression method [4] was used to compress each memory dump, one line (1024 bytes) at a time, resulting in a sequence of compressed line sizes. Any line "compressing" to greater than 1024 bytes was set to compress to exactly 1024 bytes, since as explained previously we assume that lines can be stored in an uncompressed format as indicated by a flag bit in the C-RAM directory. 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 5, 6, and 7 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 pair-wise; otherwise with an odd number all but one fragment combine pair-wise, leaving exactly one half-full fragment. Therefore in Table 5 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 since, to the accuracy used in the tables (0.1%), the compression results for best-fit are essentially the same as first-fit.

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

Table 5. Compression (%) of Memory Dumps, Block Size = 64

COHORT	AIX(active)		NT(boot)		NT(active)	
SIZE	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 = 128

COHORT	AIX(active)		NT(boot)		NT(active)	
SIZE	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

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 5 (for a block size of 64 bytes) are quite close to the "raw" line size compressions, that is, 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 case, we will see in the next section that this is not the best choice of block size when the required directory space is taken into account. Another consideration is the following: with directory entry formats as described in Section 3, each directory entry for a 1K byte line requires 16 pointers in the case of 64 byte blocks, which could have a negative impact in latency and/or complexity for the memory controller.

Unlike the results based on a uniform distribution of compressed line sizes, note that there is a significant improvement from using 3-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 non-uniformity, 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 5,828 times, i.e. they comprise approximately 9% of all lines. Due 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, in a fixed-size directory entry type 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 (as shown earlier for the line 3 example in Figure 2 of Section 3). Thus, in practice, 3-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 re-set to 0 (emulating storing the line entirely within the directory entry). The results are shown in Table 8.

COHORT	AIX(active)		NT(boot)		NT(active)	
SIZE	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

Table 8. Compression (%) of Memory Dumps, Block Size = 256
(Lines Compressing to 15 Bytes or Less in Directory)

6. On Optimal Block Sizes

In this section we consider the problem of choosing an optimal block size, given a directory structure, average compression, and effectiveness of fragment combining. The tradeoff involved in choosing a block size is the following: the larger the block size, the more wasted space there is due to fragmentation; however, the smaller the block size, the more space is required for the directory.

In order to obtain an analytic result, we will make some simplifying assumptions. One of these is that the C-RAM is operating at a fixed average compression (that is, we assume the average compression is a constant, as opposed to, say, a time varying function). As previously described, C-RAM directory structures can be classified as static and dynamic. In the case of a static 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. In order to emphasize this, we will use the term "object" to refer to a variable size entity that is 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 are using 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. Note that the collection of all such pointers could be stored contiguosly in a separate memory area, since in this simple design there is a one-to-one correspondence between blocks and pointers; however, even if each pointer were stored immediately after its associated block, the collection of all such pointers is still considered to be the directory.

Given a directory structure, 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 Fig. 2, each fixed size directory entry has 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 larger than 2 (due to fragment combining), the total number of blocks is half the total number of pointers in the directory. Therefore, in this case, q would be $q'/2$. If, for example, each fixed size directory entry were 16 bytes long, then 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 easily estimated given a particular directory structure, together with (in some cases) the assumed constant average compression. Therefore, we will 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 is not 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 again the expected size of a fragment is $1/2$ of a block. However, now this fragment is shared between two objects, so in this case the expected number of blocks to store a compressed object is $p/b + 1/4$. Given typical granule sizes and block sizes as in previous sections, in practice results would not be as good as the continuous case since distributions are discrete: for example, with 32 byte granules and 256 byte blocks, assuming a uniform fragment distribution, fragments consisting of 1, 2, ..., 8 granules are equally likely, giving an expected fragment size of 0.5625 blocks rather than 0.5 blocks. We will parameterize the effect of fragment combining as follows: given a particular fragment combining scheme, then 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 Tables 1 or 2 of Section 4), that is better than two-way fragment *catenation* assuming a uniform continuous fragment distribution, which is in turn better than two-way fragment catenation in the (more realistic) uniform discrete case.

Given the total memory size M, as discussed at the beginning of this section 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. Using a continuous approximation, an estimate of the optimal value of b can be found as follows.

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.

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

Multiplying by the constant $1/(Mn)$, differentiating, and setting the result to 0, we find the optimal value of b must satisfy:

$$\frac{1}{Mn}T'(b) =$$
$$(b+q)^{-1}(np+b)^{-1} - b(b+q)^{-2}(np+b)^{-1} - b(b+q)^{-1}(np+b)^{-2} = 0.$$

Simplifying:

$$\begin{aligned} (b+q)(np+b) &= b(np+b) + b(b+q); \\ npq &= b^2\ . \end{aligned}$$

We therefore have the following result:

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

As an example, suppose we use a directory structure for a C-RAM with fixed size 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 of fixed size directory entries 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 or best fit with granules of size 32 bytes), we find:

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

In practice, q (the directory overhead per block) is not independent of M and b (that is, it is not a constant, as assumed above), since for sufficiently large values of M (or sufficiently small values of b), it may be found that an initial estimate of the required size of pointer fields in the directory is too low. This does not seem to be a practical problem, for the following reasons. First, the number of bits required for pointer fields increases only logarithmically with the total number of blocks, whereas in actual directory designs it is desirable for performance reasons to have the directory entries containing the pointer fields aligned on at least word boundaries. Second, it is also desirable to have the block size a power of 2; however, the curve for $T(b)$ is fairly flat at $b = b_{\mathrm{opt}}$ (see Figure 3), so that picking a block size so that it is the power of 2 closest to the computed value for b_{opt} generally gives results reasonably close to the optimal value.

Note that the above can be re-written as $M/T = (b + q)(np + b)/nb$. M/T is 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, this means that 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 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? Assuming, as in the examples of Section 3, 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 in the following straightforward way. Each megabyte of logical memory consists of 1024 1K lines, which requires 64K, 32K, and 16K bytes 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 9.

BLOCK SIZE	AIX(active)	NT(boot)	NT(active)
256	436.9K	581.2K	459.4K
128	406.8K	555.3K	427.3K
64	423.4K	576.0K	446.0K

Table 9. Required Total Memory for Each MB of Logical Memory

Thus we see that even taking into account the fact that larger directory entries can provide more space to store highly compressible lines, a block size of 128 bytes (as estimated analytically above) is optimal among the cases where blocks sizes are constrained to be powers of 2. As one moves from a 256 to 128 byte block size, 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 have 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 call the *net compression*. To distinguish this from our earlier definition, we will 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 9 is shown in Table 10, in which the net compressions are found from the values given in Table 9 by dividing by 1024K. We see that 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.

BLOCK	AIX(active)		NT(boot)		NT(active)	
SIZE	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

Table 10. Comparison of Data Compression and Net Compression (%)

Similar results can be obtained for static directory designs. In a static directory design, the amount of space used by the directory is a fixed fraction of total memory determined by the maximum compression supported (a design parameter). 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).

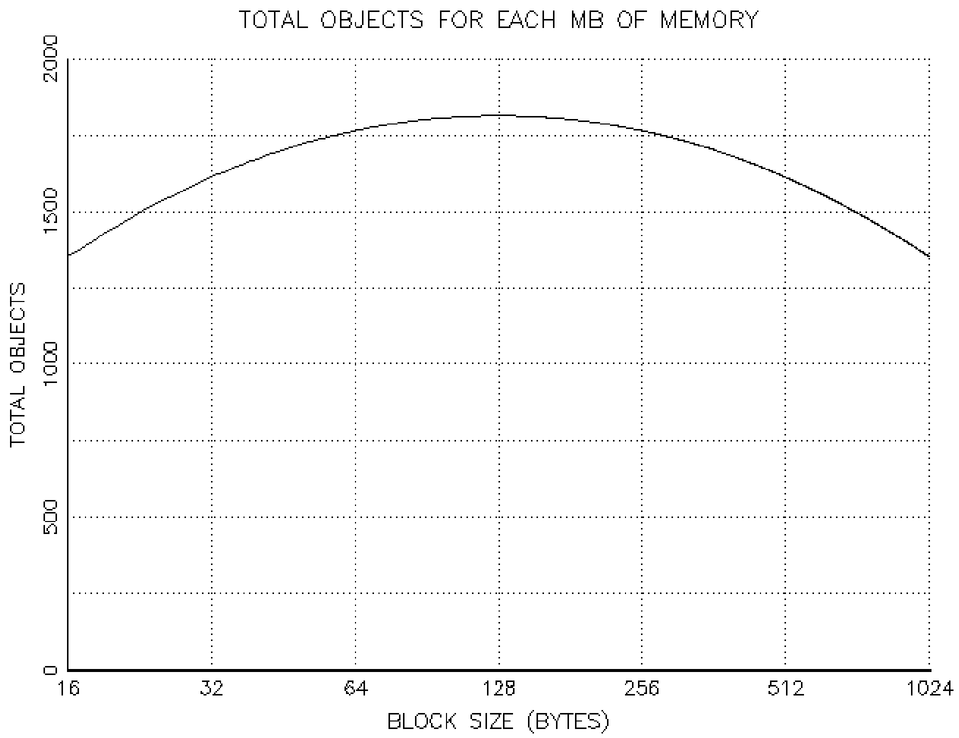

Figure 3. $T(b)$ for $M = 1$ MB, $p = 512$, $q = 8$, $n = 4$
(as predicted analytically)

7. Conclusion

Effective use of compressed random access memories at the lowest level of a system's main memory hierarchy involves finding relatively simple designs, suitable for hardware implementation, for directory structures and memory management, that 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 four, for example.

With respect to directory structure, there are two approaches: in a static design, the directory is pre-allocated 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 de-allocated 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 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 4K byte pages and 1K byte 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 can be 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, it is found that the optimal block size is on 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.

In summary, providing that compression/decompression latencies are acceptable, there appear to be no fundamental problems in using compressed random access memories at the lowest level of a system's main memory hierarchy. Although the simple method of allocating a number of fixed size blocks to store variable size compressed lines results in significant fragmentation for practical block sizes, extending this so that pairs of fragments from small predetermined sets of lines can be combined yields low fragmentation levels, and these methods are suitable for hardware implementation. 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. In any case, if the cost of memory in a large machine is from 50% to 90% of the total cost, reducing this part of the cost by half (or more) by means of compressed memory yields savings ranging from 25% to 45%.

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

References

[1] Pietrek, M. and Seltzer, L. Windows 3.1 memory enhancement utilities. *PC Magazine 15*, 4 (Feb. 20, 1996), 205-213.

[2] Kjelso, M., Gooch, M. and Jones, S. Design and performance of a main memory hardware data compressor. In *Proc. 22nd EUROMICRO Conf.*, pp. 423-430, IEEE, 1996.

[3] Douglis, F. The compression cache: using on line compression to extend physical memory. In *Proc. Winter 1993 USENIX Conf.*, pp. 519-529, USENIX Assoc., 1993.

[4] Franaszek, P., Robinson, J., and Thomas, J. Parallel compression with cooperative dictionary construction. In *Proc. DCC '96 Data Compression Conf.*, pp. 200-209, IEEE, 1996.

[5] Franaszek, P., Heidelberger, P., and Wazlowski, M. On management of free space in compressed memory systems. In preparation.