A Robust Main-Memory Compression Scheme

Magnus Ekman and Per Stenstrom
Department of Computer Science and Engineering
Chalmers University of Technology, SE-412 96 Goteborg, Sweden
Email: {mekman,pers}@ce.chalmers.se
Members of the HIPEAC Network of Excellence

Abstract

Lossless data compression techniques can potentially free up more than 50% of the memory resources. However, previously proposed schemes suffer from high access costs.

The proposed main-memory compression scheme practically eliminates performance losses of previous schemes by exploiting a simple and yet effective compression scheme, a highly-efficient structure for locating a compressed block in memory, and a hierarchical memory layout that allows compressibility of blocks to vary with a low fragmentation overhead. We have evaluated an embodiment of the proposed scheme in detail using 14 integer and floating point applications from the SPEC2000 suite along with two server applications and we show that the scheme robustly frees up 30% of the memory resources, on average, with a negligible impact on the performance of only 0.2% on average.

1. Introduction

As the processor clock speed has increased, the gap between processor and memory and disk speed has widened. Apart from advances in cache hierarchies, computer architects have addressed this speed gap mainly in a brute force manner by simply wasting memory resources. As a result, the size of caches and the amount of main memory, especially in server systems, has increased steadily over the last decades. Clearly, techniques that can use memory resources effectively are of increasing importance to bring down the cost, power dissipation, and space.

Lossless data compression techniques have the potential to utilize main-memory resources more effectively. It is known from many independent studies (e.g., [1][11]) that dictionary-based methods, such as LZ-variants [15], can free up more than 50% of main-memory resources. Unfortunately, to manage compressed data with no performance losses is quite challenging and presumably the reason it has not yet seen widespread use: Firstly, since decompression latency is on the critical memory access path, it must be kept low. Secondly, since size of compressed data can vary, the mapping between the logical and the compressed address space is non-linear and the translation between the two spaces also ends up on the critical memory-access path and must be efficient. Finally, since compressed blocks have different sizes, fragmentation may reduce the amount of memory freed up. How to efficiently keep fragmentation overhead at a low level is therefore another key issue.

Due to the significant decompression latencies of early schemes for memory compression [6][12][10][21][4], compression was only applied to pages that were victimized. The memory freed up was used to bring down paging overhead. More recently, Abali *et al*.[1] disclosed the IBM MXT technology [19][20]. Unlike previous proposals, the entire memory is compressed using an LZ-based compression algorithm [20]. On each memory access, the compressed block has to be located through a translation that involves an indirection through a memory-resident table. This effectively doubles the memory access latency. The located block is then decompressed which, despite of a parallel implementation using a special-purpose ASIC, takes 64 cycles [20]. To shield the processors from these devastating latencies, a large (32 Mbyte) tertiary cache in front of the compressed memory is used. If the working-set size exceeds this cache capacity, however, the increased latencies may cause significant performance losses.

In this paper, we propose a novel main-memory compression scheme that frees up a significant portion of the memory resources *at virtually no performance cost* by addressing the three critical performance issues above. First, to get a negligible decompression latency, we use a computationally lightweight compression algorithm. We have leveraged on the observation made by Yang and Gupta [22] that especially the value zero is frequently used in a program. Alameldeen and Wood [2][3] recently exploited it for cache compression in their frequent-pattern compression (FPC) technique that codes blocks, words, and bytes containing zero very efficiently. FPC is simple and can decompress a block in a handful of cycles.

Our first contribution, which extends the observations in [22], is the establishment that *'zero-aware' compression techniques* are indeed effective when applied to in-memory data. Compressibility data for 14 applications from the *SPEC2000* suite and from two server applications show that such compression techniques can potentially free up

0-7695-2270-X/05/$20.00 (C) 2005 IEEE

50% of memory — zero-aware techniques are therefore competitive with computationally more demanding dictionary-based methods as used in, *e.g.*, IBM/MXT [19][20].

The main contribution of this paper is a novel main-memory level compression scheme that can free up 30%, on average, of the memory resources with a performance overhead of only 0.2%. The route to this finding, however, made us solve several technical challenges. First, in order to locate a compressed memory block with a low overhead, our proposed memory mapping scheme allows for compactly keeping address translations in a TLB-like structure. This yields two advantages: (1) Address translation can be carried out in parallel with the L2 cache and (2) Blocks containing only zeros are quickly detected. Second, our scheme brings down fragmentation caused performance overhead by initiating relocation of parts of the compressed region only occasionally. We show that relocation happens sufficiently rarely to not cause any performance losses.

We first present compressibility statistics of zero-aware as well as other well-known compression techniques in the next section. We then present our memory-level compression scheme in Section 3. In Sections 4 and 5, we evaluate the performance of our compression technique focusing on fragmentation overhead and its potential performance losses. Section 6 relates our findings to work done by others before we conclude in Section 7.

2. Compressibility of main memory

This section first establishes the relative occurrence and distribution of zero-valued locations in memory-level data. While Yang and Gupta [22] established that words containing zero are very common, we extend their results by looking at distributions across entities spanning from a single byte to a whole page. In contrast, Alameldeen and Wood [2][3] considered the relative occurrence of zero-valued locations at the cache-level and we will see whether their results extend to in-memory data. This is the topic of Section 2.1. Then in Section 2.2, we compare the compressibility of a class of zero-aware compression algorithms to more demanding dictionary-based methods. The results in this section are based on measurements on images that were created at the end of the run of the benchmarks. Measurements in Section 5 will confirm that this is representative data by measuring several points throughout the execution of the benchmarks.

2.1. Frequency and distribution of zero-values

Figure 1 shows the percentage of consecutive zeros and ones in the memory footprints for 16 benchmarks (see Section 4) on a SPARC based machine. For each application, the first bar shows the fraction of 8-Kbyte pages that are zero. The second, third, and fourth bars show the fraction of 64-byte blocks, 4-byte words and single bytes that are zero, respectively. The fifth and sixth bars show the fraction of blocks and single bytes that only contain ones.

As seen by the rightmost set of bars, which average the statistics across all applications, as many as 30% of the 64-byte blocks only contain zeros. Further, 55% of all bytes in the memory are all zero! The fraction of bytes and blocks only containing ones is low for most of the benchmarks but significant for two of them (*mesa* and *SpecJBB*) which suggests that they could be a target for compression. This data clearly shows that zero-aware (and possibly one-aware) compression techniques with byte-granularity have a great potential also at the memory level. We next compare their compressibility with more effective, but more computationally demanding, techniques.

2.2. Compressibility with zero-aware algorithms

The zero-aware techniques we consider are variations of Frequent Pattern Compression (FPC) [3]. In the baseline technique (from [3]), each word is coded by a three-bit prefix followed by a variable number of additional bits. The three-bit prefix can represent eight different patterns of which one represents uncompressed data and is followed by four bytes in uncompressed form. The seven other patterns are, zero-run (up to eight consecutive words are zero), 4-bit sign extended (SE), single byte SE, halfword SE, halfword padded with zero halfword, two halfwords each consisting of a sign-extended byte, and finally a word consisting of repeated bytes. The bits used to sign-extend 4-bit, byte, and halfword operands are thus compressed into a 3-bit prefix plus the operand.

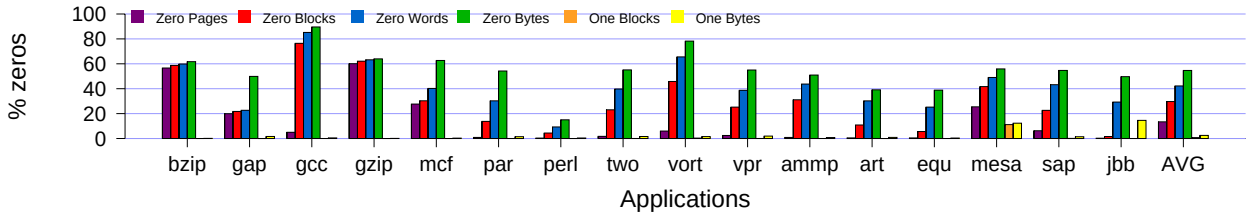

FIGURE 1. The graph shows the fraction of memory that contains zeros when partitioned into pages, blocks, words, and bytes corresponding to the first four bars. from left to right The fifth and the sixth bars show the fraction of blocks and single bytes that only contain ones.

Figure 1 suggests that negative numbers are not expected to be frequent since there are few occurrences of runs of consecutive *ones*. Further, it is intuitive to believe that some of the patterns supported by FPC are not very frequent either. We thus evaluate three simpler variations of FPC. The first one — *FPC only zeros* — uses eight patterns but does not code sign-extended negative numbers. The second one — *FPC simple* — only uses two prefix bits and thus only has four patterns (zero run, 1 byte SE, half-word SE, and uncompressed). Finally, the third variation — FPC simple only zeros — uses four patterns but does not allow sign-extended negative numbers.

We compare the compressibility of these FPC variations with some more computationally demanding compression techniques: X-MatchPro[14], which is designed to be implemented in hardware; and LZSS [17] which is similar to what is used in [1] and recently was proposed to be used for caches [8]. These algorithms are applied to 64-byte blocks. Further, to get a practical bound on the compressibility of the memory footprints we used the deflation algorithm (a combination of LZ77 [24] and static Huffman coding [9]) in zip and gzip on the entire footprints.

Figure 2 shows the compressibility of the various algorithms. The rightmost group of bars shows the average compressibility across all benchmarks. The first four bars in this group shows that the original version of FPC provides rather small advantages over the simplified versions, but for some of the benchmarks (*gap*, *mesa*, *SAP* and *SpecJBB*) there are significant differences. Further, LZSS and X-MatchPro do better than FPC but the difference is not very large. While the deflate algorithm outperforms FPC with more than a factor of two, it is important to remember that deflate was applied to the entire image to establish a practical bound on compression; the other algorithms were applied to individual 64-byte blocks.

The graph shows that it is possible to achieve about 50% compression with zero-aware compression techniques such as FPC and in the compression scheme described in the next section, we assume the FPC although the particular choice of algorithm is orthogonal to our scheme.

3. Proposed memory compression scheme

An overview of the proposed architecture is shown to the left in Figure 3. Data is compressed when it is brought in from disk so that the entire main memory is compressed. Each block is compressed individually using the baseline FPC algorithm detailed in Section 2 and is assigned one out of n fixed sizes after compression. These sizes are called *block thresholds* and are denoted t_0, t_1,..., t_n. The first and the last block thresholds are special in that they correspond to a block containing only zeros (t_0) and an uncompressed block (t_n). Decompression of a memory block is triggered by a miss in the lowest level of the *cache* hierarchy. This may cause a block eviction and if the evicted cache block is dirty it needs to be compressed before it is written back to memory.

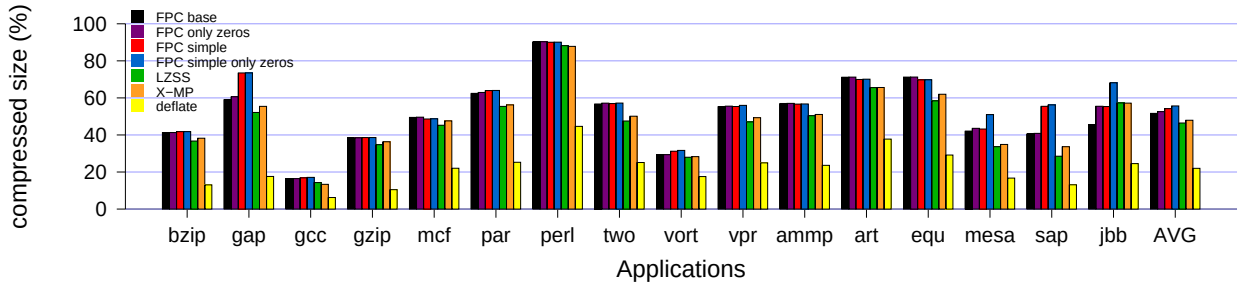

FIGURE 2. Algorithm comparison. The bars represent resulting size (in %) for the seven algorithms.

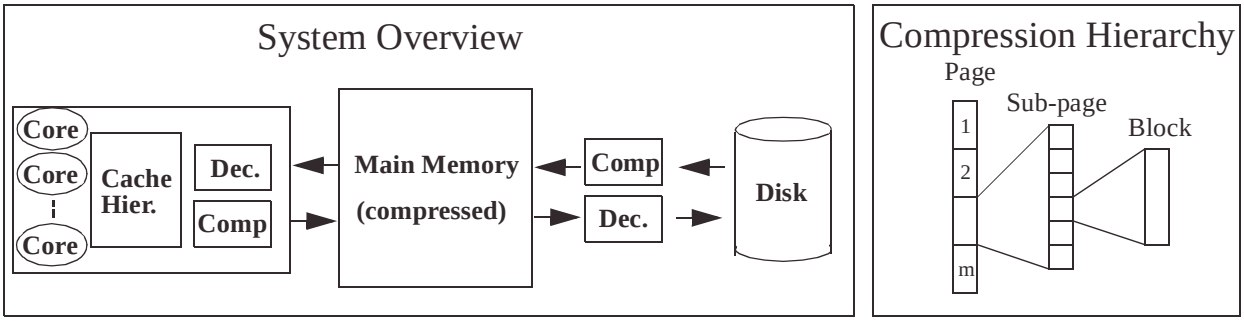

FIGURE 3. Left: Overview of the compressed memory architecture. Right: Compression hierarchy partitioned into pages, sub-pages, and blocks.

If the size of the dirty block now matches another block threshold, the block will be assigned this new block threshold. Then, consecutive blocks in the address space need to be moved to make room for the block in the case the block size increases, denoted a *block overflow*, or to fill up the empty space in the case the block size decreases, denoted *block underflow*.

To reduce the number of occasions when blocks need to be relocated, a small *hysteresis*, h_k, is introduced for each threshold so that a block needs to be h_k bytes smaller than threshold t_k to change size. Thus, a block can always grow slightly without overflowing and once it has overflowed, its size needs to be reduced h_k bytes before a new relocation is needed. Further, to reduce the amount of data that needs to be relocated, each page is divided into *m* sub-pages in a hierarchical manner as shown to the right in Figure 3. Assuming such a hierarchy, only the blocks located after that block within the sub-page need to be relocated on a block overflow/underflow. The sub-pages can also be one out of *n* sizes called *sub-page thresholds* and the same holds for the pages. Hystereses are used at each level to reduce the number of sub-page and page overflows/underflows.

3.1. An example system

Figure 4 shows one instantiation of the proposed compressed memory system to serve as an example. The number of block thresholds (*n*) in this example is four and they are set to 0, 22, 44, and 64, assuming a block size of 64 bytes. As we will show experimentally, setting t_0 to zero takes advantage of the observation that many cache blocks only contain zeros. The threshold of each block is encoded with $log_2 n$ bits that are kept as a bit vector called *size vector* in the page table together with the address translation. Assuming four thresholds, the block sizes can be encoded with two bits as the values 00, 01, 10 and 11, where 00 denotes a block with only zeros, and 11 denotes an uncompressed block. The size of each sub-page, as well as the size of the page itself could be calculated from these bits, but to simplify the address translation process, the size vector also contains the encoded sizes for the sub-pages and the page.

The lower left part of Figure 4 shows the beginning of a page in memory and three blocks. Above that, the corresponding size vector (11 01 00 10....), is shown. The compressed size of block 0 is between 44 and 64 and hence is assigned size 64 (11). Block 1 is less than 22 and is assigned the size 22 (01). Block 2 only contains zeros and since size (00) is reserved for zeros, this block does not have to be stored in memory at all, given that the corresponding *size vector* is stored elsewhere. Thus, after block 1, block 3 is located, with the size code (10).

The size vector is needed on a cache miss to locate the requested block. To avoid requesting the size vector from memory, our scheme uses a structure on the processor chip called Block Size Table (BST) that caches block size vectors. This table has one entry per TLB entry and thus have the same miss rate and is filled on TLB-misses by the TLB-miss handler.

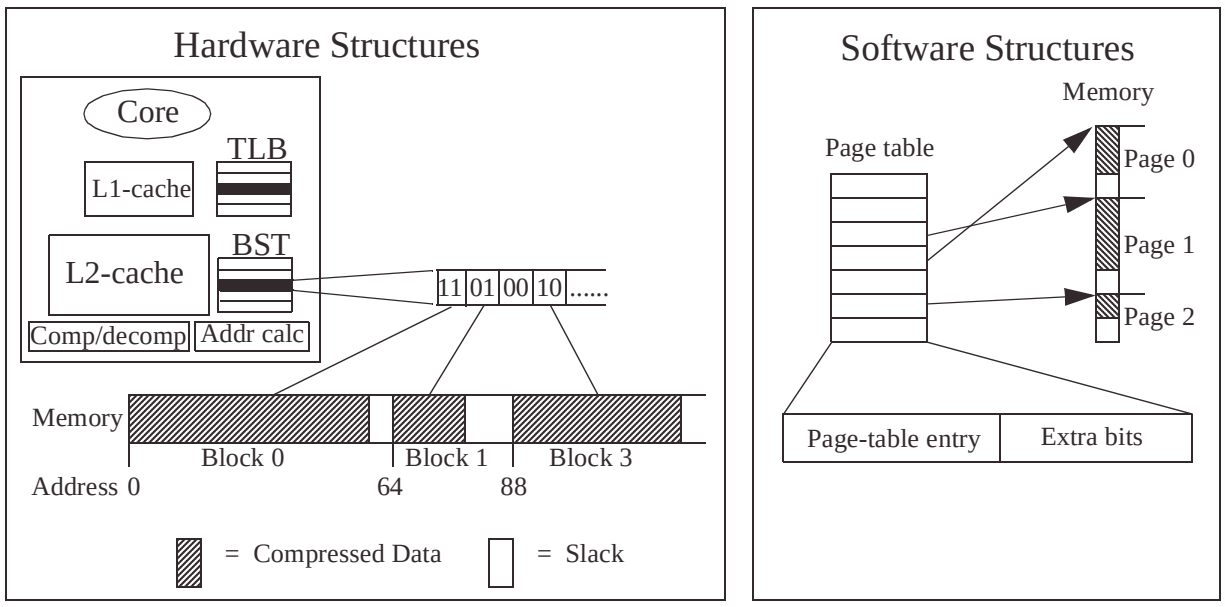

FIGURE 4. Proposed architecture. Left: processor chip with the new structures added. Right: page table where each entry contains extra address bits since the page can be of a smaller size than the baseline system as well as bits for each block in the page encoding the current compressed size.

Each entry in the BST consists of $log_2 n$ bits for each memory block, sub-page, and the page itself in the corresponding page so the total number of bits for each entry is $(PAGE\_SIZE/BLOCK\_SIZE+m+1)*log_2 n$, where *m is* the number of sub-pages in a page. In our example, assuming a page size of 8 KB, eight sub-pages, and a cache block size of 64 bytes, each entry consists of 34.25 bytes. A TLB-entry that contains one physical and one virtual address, is typically 16 bytes in a 64-bit system, and hence the BST is about twice the size of a TLB. In the example the BST is located and accessed in parallel with the L2-cache but an alternative location would be on the memory controller. Having it on the processor chip comes with an immediate advantage though. If a block with only zeros is requested, the memory request can be immediately satisfied on the processor chip since the size vector indicates zero. Thus, in case of a load, a zero is immediately returned and the load instruction is completed quickly. The corresponding cache block is then cleared in the background, in parallel.

As mentioned above, the BST entry also contains the encoded sizes of the sub-pages in the page. To locate the address of a compressed block, the offset within the page must first be determined. This is done in the *address calculator* by adding the sizes of the sub-pages prior to the block and the sizes of the blocks located prior to the requested block in the same sub-page. To be able to do this, the address calculator keeps a table of the block thresholds and sub-page thresholds which are set globally for the system.

The address calculator must in the worst case add $m-1$ sub-page sizes and $PAGE\_SIZE/(m*BLOCK\_SIZE)-1$ block sizes. With the previously assumed parameters and eight sub-pages for each page, this yields an addition of at most 22 small integers of between 6-10 bits. Another solution would be to first count the number of blocks for each of the *n* sizes and then multiply this with the size, and then do the same for the sub-pages and then add these two results together. We did preliminary timing and area estimates of this logic and found out that such a calculator that could complete the task in six cycles would at most occupy the space of between two and three 64 bit adders, which is negligible for modern processors. Given a BST access time of about twice the TLB access time and an address calculation of only six cycles it should be possible to hide this latency with the L2-cache access. Thus, in the case of an L2-cache miss, we already know where to find the block and no extra latency is added to the miss handling time. Power dissipation of the BST is not an issue if its size is comparable with a TLB and only accessed on L1-misses.

3.2. Block overflow/underflow handling

On a write-back, the block is first compressed and the size is checked against the bits in the size vector. In case of a block overflow or underflow, data in that sub-page need to be moved. The actual move is taken care of by an off-chip DMA-engine in the background so that the processor can continue executing and avoid to consume precious chip bandwidth. We assume that the DMA-engine uses cycle-stealing and does not exclude the processor from accessing memory that is not on the same sub-page or page that is being moved. When the transfer is done, the BST entry needs to be updated (by hardware) to reflect the new block and sub-page sizes. Later, this entry needs to be written back to the page table if it is not updated at the same time as the BST entry. We note that these updates raise issues similar to TLB consistency [18] in a large scale system but do not elaborate on it more in this paper.

When data is being moved, no further misses to that sub-page (or page in the case of a sub-page overflow/underflow) can be handled and the processor has to stall on such misses. However, since overflows can only happen on a write-back which is expected to happen when the block has not been used for a while, it is reasonable to believe that blocks located close to the block will not be needed in the near future due to spatial locality. Thus block/subpage moves should not affect performance by much. In the rare event of a page overflow, a trap is generated and the virtual memory manager will have to find a new location for the page and notify the DMA-engine to perform the move. We will experimentally verify that overflows/underflows do not introduce any severe performance losses.

3.3. Operating-system modifications

As mentioned before, the page table must be modified to include the size vectors so that they can be loaded just like the address translation on a TLB-miss. Also, since the pages are of variable size, pointers to pages must use more bits. Assuming that the page thresholds must be of a size that is a multiple of *p* bytes, the number of extra bits will be $log_2(PAGE\_SIZE/p)$. Together with the size vector the storage overhead will be $(PAGE\_SIZE/BLOCK\_SIZE)*log_2 n + log_2(PAGE\_SIZE/p)$ per page in the pathological case when no gain is made from compression. If we assume that all of the page thresholds are multiples of 512 bytes, this storage overhead is a tiny 0.4%.

The virtual memory manager also needs to be slightly modified so that instead of just keeping one pool of free pages, it needs to maintain one pool for each of the *n* compressed page sizes. On a page fault, the page is read from disk, compressed and assigned a page size. The size vector is also created at this point and inserted into the page table. Also, if the page table entry is not updated on each overflow/underflow, the TLB-miss handler needs to write back the evicted BST entry on a TLB miss so the page table contains the correct size vector.

4. Experimental methodology

To evaluate the proposed memory compression scheme we have used simulation at two levels of detail: one *cycle accurate* and one simpler but much *faster* that models the processor as an in-order single issue and has the advantage that it is fast enough to enable us to run the benchmarks to completion. Both simulators are based on Simics [13] which comes in various flavors with various levels of detail. The fast simulator uses the timing-model interface for the in-order version of Simics while the cycle-accurate simulator uses the Micro-Architectural Interface (MAI) version of Simics and models a dynamically scheduled multiple-issue out-of-order processor with non-blocking caches. For our multithreaded workloads we simulate a chip multiprocessor with several copies of the same processor core but with a shared L2-cache where the size is scaled by the number of threads.

The fast simulator is used to collect statistics about compression rate and overflows/underflows to establish the wins with compression as well as building intuition into performance-related issues while the cycle-accurate simulator is used to get detailed results about impact on the number of executed instructions per second (IPC). Since the cycle accurate simulator is too slow to run the entire benchmarks we have simulated a single simulation point for each benchmark that is representative of the overall behavior of the benchmark according to [16]. Table 1 shows the baseline parameters for the simulated system. The instruction set architecture (ISA) is Sparc-V9.

The memory latency of 150 cycles is probably unrealistic[1] for a future system but this is intentionally chosen not to favor our compression scheme since we add a few cycles to the access path due to decompression which performance wise becomes worse with a short memory latency.

The last three columns in the table shows the number of cycles that sub-pages and pages are locked in the case of a block, sub-page or page overflow/underflow. We have chosen the times conservatively by assuming DDR2 SDRAM that is clocked at 200 MHz which provides a bandwidth of 3.2 GBytes/s. Streaming out an entire 8K page will thus take 2.56 microseconds. Writing it back to a different location takes the same amount of time, yielding a total of 5.12 microseconds. Assuming a processor frequency of 4 GHz[2] this would be 20 480 cycles. Moving all the blocks within a sub-page would take 2 560 cycles. We add about 1 500 cycles as an initial latency to both these figures. We note that the lockout time could be shorter at the occasions when only parts of a sub-page or page need to be moved (in the common case when the overflow does not occur for the first block in the sub-page or page) but conservatively we assume the same lockout time for all cases.

4.1. Benchmarks

The benchmarks used are summarized in Table 2. We used ten integer applications and four floating point applications from the *Spec2000* benchmark suite[3] using the *reference data set* to evaluate the compression scheme. Since Simics is a full-system simulator we were also able to simulate the two multi-threaded server workloads *SAP Sales and Distribution* and *SpecJBB*. *SAP* runs together with a full-fledged database and *SpecJBB* is a Java-based benchmark. Both these multi-threaded benchmarks run eight threads and were tuned for an 8-way UltraSparc-III system running Solaris 9. With the fast simulator the *Spec2000* benchmarks were run to completion and *SAP* and *SpecJBB* were run for 4 billion instructions on each processor during their steady-state phase (verified with hardware counters). With the cycle-accurate simulator they were all run for 100 million instructions after warming.

1. Measured latency on a 2.2 GHz AMD Opteron 248 is *142 cycles* and on a 3.4 GHz Intel Xeon64-Nocona *432 cycles*. Latencies measured with *Calibrator* (http://homepages.cwi.nl/~manegold/Calibrator/).

2. This is not consistent with the assumption of 150 cycles latency to memory but assures that we are conservative with respect to lock times and thereby do not give our proposed scheme and advantage.

3. Two integer benchmarks were left out due to compilation and running problems, and only four floating point benchmarks were used to reduce the total number of applications.

TABLE 1. Parameters for the simulated system.

Parameter	Instr. Issue	Instr. Fetch	Exec. units	Branch pred.	L1 i-cache	L1 d-cache	L2 d-cache	Mem-latency	Block lock	Subpage lock	Page lock
Value	4-way out of order	4	4 int, 2 int mul/div, 2FP, 1FP mul/div	16k-entry gshare, 2k-entry BTB, 8 entry RAS	32k, 2-way, 2-cycle	32k, 4-way, 2 cycle	512k/thread, 8-way, 16 cycle	150 cycles	4000 cycles	23000 cycles	23000 cycles

TABLE 2. Benchmarks

Integer	bzip	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vpr
FP and server	ammp	art	equake	mesa	SAP S&D	SpecJBB			

5. Experimental results

This section presents the experimental results. We start by investigating how much of the potential compressibility that is lost due to fragmentation depending on how the block, sub-page and page thresholds are chosen in Section 5.1. We then investigate the performance critical parameters in Section 5.2 and present detailed performance results in Section 5.3.

5.1. Fragmentation and thresholds

The compression results presented in Section 2.2 assumed that a block can be of an arbitrary size but often this is not practically implementable. In the architecture presented in Section 3, it is assumed that a block can be any of *n* sizes where *n* is chosen to be four in our evaluation. Further, a number of blocks are clustered into a sub-page that also can be one of four different sizes (called sub-page thresholds). Finally, the sub-pages make up a page that also can be one of four sizes (called page thresholds). Each of these levels in the hierarchy will cause fragmentation which reduces the gains from compression. The most straight-forward way to select the threshold values is to place one of them at the maximum size and place the rest of them equi-distantly between the maximum value and zero as shown in Table 3. Note that the way that the block thresholds are chosen is slightly different, in the sense that the first block threshold is chosen to be zero. The effects of this will be studied later.

TABLE 3. Threshold values.

	1	**2**	**3**	**4**
block	0	22	44	64
sub-page	256	512	768	1024
page	2048	4096	6144	8192

The resulting fragmentation is shown in Figure 5 which shows the resulting size in percent compared to the original size. This ratio was measured every 1-billion instructions and the bar represents the mean value. In each point, this ratio was calculated for all pages that had been touched at that time. For the *Spec2000* applications, only user accesses were considered to avoid measuring operating system pages that are typically shared across applications since this would bias our results. Tests with system accesses did not change the results significantly. The left-most bar for each benchmark represents the compressed size without fragmentation. The next three bars represent the achieved compression when fragmentation is taken into account at each level. The rightmost bar is the most interesting one since that includes fragmentation from all levels and thus represents the resulting size after losses due to fragmentation. Despite the losses due to fragmentation there are still great wins to be made by using compression — 30% of the memory is freed up!

Apart from that the compression gains are large, the compressibilities of most benchmarks are rather stable over time. This is shown in Table 4 which contains the standard deviation of the compression ratio where the compression ratio is measured in percent. Of all the *16* benchmarks only *gap*, *gcc* and *mcf* have a standard deviation that is higher than *10*.

As described in the previous section the thresholds for the blocks, sub-pages and pages were chosen equi-distantly. This is the best choice if the block sizes are uniformly distributed. If the distribution is not uniform, however, so that many blocks are of one or a few sizes close to each other, it will be advantageous to choose the threshold values to be just slightly above the sizes for these blocks, since that will reduce the losses due to fragmentation.

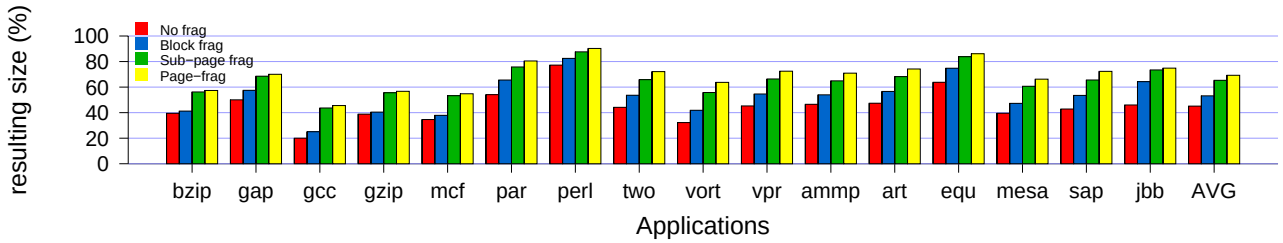

FIGURE 5. Mean size with equi-distant thresholds. The left most bar contains no fragmentation. For the other bars, fragmentation on the three levels (block, sub-page and page) is added.

TABLE 4. Standard deviation of the compression.

bzip	**gap**	**gcc**	**gzip**	**mcf**	**par**	**perl**	**two**	**vort**	**vpr**	**am**	**art**	**equ**	**mes**	**sap**	**jbb**
5.5	16.9	12.4	5.5	13.7	0.97	1.97	0.83	2.2	0.93	1.64	1.79	1.19	0.80	2.39	0.93

Actually, a first step of exploiting this was already done in the above experiment when reserving one of the block sizes for blocks with only zeros. We will next consider the effects of not assigning one size to zero as well as choosing the thresholds more specifically to reduce the fragmentation.

We have investigated fragmentation for five different approaches to establish the thresholds. They can be found in Table 5 and the resulting sizes are shown in Figure 6.

TABLE 5. Summary of investigated sets of thresholds.

	Block thresh.	Sub-page thresh.	Page thresh.
Equi	16, 32, 48, 64	256, 512, 768, 1024	2048, 4096, 6144, 8192
Equi zero	0, 22, 44, 64	256, 512, 768, 1024	2048, 4096, 6144, 8192
System global	0, 22, 44, 64	22, 400, 710, 1024	1024, 4096, 6144, 8192
Benchmark global	optimized per benchmark	optimized per benchmark	1024, 4096, 6144, 8192
Benchmark local	optimized per page in each benchmark	optimized per page in each benchmark	512, 4096, 5632, 8192

The first one has equi-distantly selected thresholds. The second one is the one already evaluated with one size assigned to zero. The third set of thresholds are calculated to optimize for the mean value of all benchmarks (given the distribution of compressed sizes for the various benchmarks). Note that the block thresholds happened to be the same as the once in the previous case but the sub-page and page thresholds differ. The fourth set contains individual block and sub-page thresholds for each benchmark, but global page thresholds. Such thresholds could be used in a system where the thresholds can be included in the binary after profiling an application. The final set of thresholds consists of individual block and sub-page thresholds for each page in each benchmark. This is the fragmentation that could be achieved by an adaptive mechanism that adjusts the thresholds for each page at run time.

For some of the benchmarks (*e.g., gcc*) the chosen thresholds cause significant variations in the resulting size while for others (*e.g., equake*) the variations are fairly small. The reason that *gcc* benefits significantly from other individual thresholds is that the size distribution of compressed blocks is not uniform but very skewed.

Also note that for three applications (*parser*, *equake* and *SAP*) the thresholds that are optimized for all benchmarks imply more fragmentation than equi-distant thresholds. It comes naturally from that we optimize for the average of all applications that some of them will suffer. This average is represented by the rightmost group of bars. According to this average, it is slightly better (with respect to fragmentation) to treat zeros in a special way (second bar) compared with not doing it (leftmost bar). Further, it is possible to reduce the amount of memory resources another few percentage units by choosing thresholds that are optimized for the distribution of the entire system. If individual thresholds (only individual block and sub-page thresholds since the page-thresholds need to be common for all benchmarks) are allowed for the different benchmarks, memory resources can be slightly reduced. However, the big wins are when individual thresholds are allowed for each individual page. This reduces the size from 67% to 60%. For the rest of this paper we use globally optimized thresholds for the system (third bar) but note that an extension to this work could be to find an adaptive mechanism that adjusts the thresholds for each page to approach the size represented by the right most bars.

As described in Section 3, it is possible to use a small hystereses to reduce the number of overflows/underflows. Further, to reduce the bandwidth consumption in the memory system we also propose to create a hierarchy by introducing the unit sub-pages. Using these mechanisms comes with a price in terms of increased fragmentation however. Experiments with and without sub-pages and hystereses resulted in that with no sub-pages the size would be 63% instead of 67%, and if both sub-pages and hystereses are used the size becomes 69%.

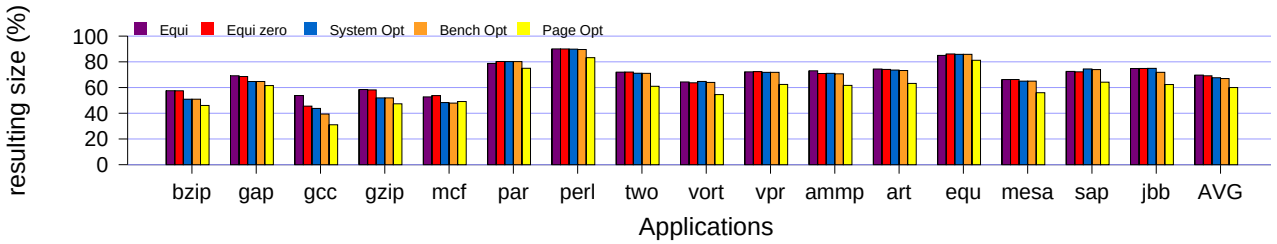

FIGURE 6. Mean compression size (including fragmentation) for various compression thresholds. No hystereses is used.

5.2. Performance critical parameters

Now that the compressibility and losses due to fragmentation have been established we will move over to the performance critical issues. With our proposed architecture, one of the three performance critical parameters have been virtually eliminated with the introduction of the BST. The two others, related to decompression latency and relocation of data will be studied next. We start with the decompression latency. The results in this section are based on the *fast* simulator using *complete* runs of the Spec2000 benchmarks, and *long* runs of the server benchmarks as described in Section 4. Cycle accurate performance results based on shorter runs of representative simulation points [16] will be presented in Section 5.3.

5.2.1. Decompression latency and zeroed blocks. First of all, let us consider the decompression latency. With a decompression latency of *5* cycles [3], and a memory latency of 150 cycles decompression will increase the memory latency by 3%. A coarse-grained estimate will thus be that for memory-bound applications that spend most of the time in the memory system, the decompression latency could degrade performance by at most *3%*. In reality, the degradation will be lower because of the latency-tolerance of the processor and will be investigated in detail with cycle-accurate simulation in Section 5.3. The effect of reading zeroed blocks could be more significant however, since each access to a zeroed block will remove one memory access. Moreover, given that the processor is not able to hide the entire memory latency, it could boost the performance in some cases. Table 6 shows the percentage of the L2-cache misses that access blocks that only contain zeros for two cache sizes (512 KB and 4MB per thread).

For many applications, only a small amount of the L2-misses access zeroed blocks but for some applications, such as *gap*, *gcc*, *gzip* and *mesa* more than *20%* of the misses access zeroed blocks. Thus, the effect of reading zeroed blocks could improve performance for a few of the applications depending on how much time they spend in the memory system.

5.2.2. Performance effects of overflows and underflows. The third performance critical issue, which is how relocation of data on overflows/underflows is handled depends on the behavior of the benchmarks. Intuition says that in the common case, locking data that will be relocated should not affect performance by much since they are contiguously located to a block that recently have been evicted from the cache and thus should not be accessed in the near future. We will now test this hypothesis. We focus on the system with a 512-KB cache per thread since this increases the miss-rate considerably for some applications and thus makes the number of evictions, and thereby possibly overflows/underflows more frequent. Larger caches should only reduce this potential performance problem.

The bars of Figure 7 show the number of overflows/underflows per committed instruction for all applications. The leftmost bar for each benchmark represents overflows/underflows for blocks, followed by sub-pages for the middle bar and pages for the rightmost bar. Note that the y-axis is logarithmic.

TABLE 6. Percent of L2-cache misses that access a zero block for two different cache sizes. First row: 512K L2-cache per thread. Second row: 4MB L2-cache per thread.

	bzip	gap	gcc	gzip	mcf	par	perl	two	vort	vpr	am	art	equ	mes	sap	jbb
512K	3.0	27.3	24.8	33.3	0.07	5.9	18.8	0.24	12.1	6.0	0.55	0.05	0.07	45.3	12.6	2.4
4MB	15.9	29.3	67.1	42.1	0.55	0.03	27.3	0.25	23.1	40.0	10.3	20.0	0.07	53.3	14.5	1.1

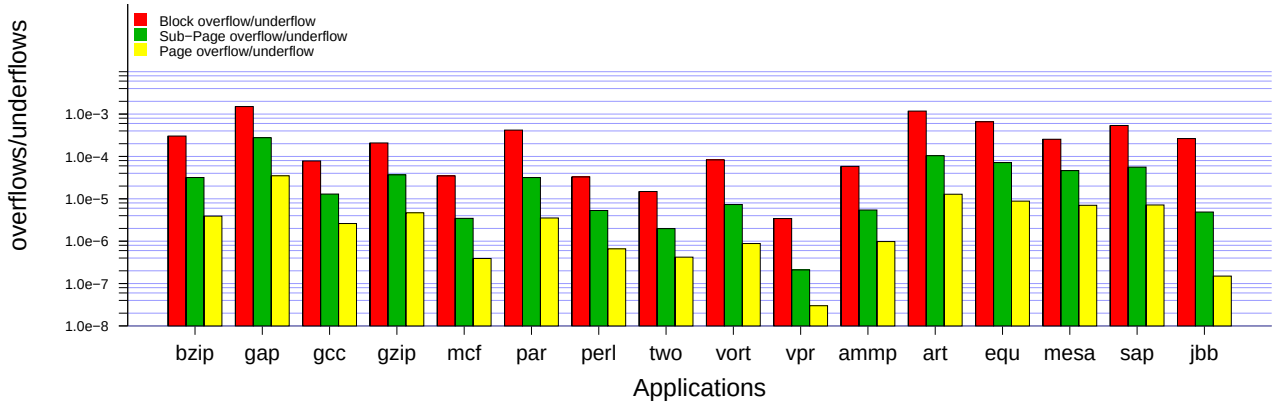

FIGURE 7. Number of overflows/underflows per committed instruction. Bars from left to right: block, sub-pages, pages. The y-axis is logarithmic.

The graph shows that for each level in the hierarchy, the number of overflows/underflows are reduced by about an order of magnitude. The main result from this graph is that it justifies the use of sub-pages, since this will greatly reduce the memory bandwidth consumed by data relocation. If sub-pages were not used, then each block overflow/underflow would trigger the relocation of half a page of data on average (assuming that the block overflows/underflows are uniformly distributed across the page). With the use of sub-pages, only half a sub-page of data needs to be relocated in about 90% of these occasions.

The bandwidth effects of the above events turn out to be that the peak bandwidth is virtually unchanged if sub-pages are used since the number of overflows for *mcf*, which is the most bandwidth demanding application, is negligible compared to the number of cache misses. If sub-pages would not have been used, the peak bandwidth had increased by 77% making *art* the most bandwidth-consuming application. The average bandwidth on the other hand increases by 58% with sub-pages and 212% without sub-pages. The average bandwidth is of more interest for multi-threaded systems (SMT, CMP or multiple processor chips). However, note that the discussed bandwidth is the memory bandwidth and not the processor chip bandwidth since an off-chip DMA-engine performs the data relocation.

While the number of overflows/underflows affects the bandwidth pressure to memory, it does not have a first order effect on performance since the processor can continue to execute while the DMA-engine is relocating data. If the processor tries to access data in memory that is locked, however, it does have to stall (referred to as *lock stall*). Intuition says that the number of lock stalls depends on for how long the data is locked. We have experimented with three relocation latencies each for blocks, sub-pages and pages. They are 100, 1000, and 10 000 cycles for blocks, 1000, 10 000, and 100 000 for sub-pages and 10 000, 100 000, and 1 000 000 for pages. The ranges include the chosen relocation latencies of 4000, 23000, and 23000.

The main results were that the number of block lock-stalls are magnitudes lower than the number of block overflows/underflows. For example, according to Figure 7, *gzip* has slightly more than 10^{-4} block overflows/underflows per committed instruction. However, even if the sub-page would be locked for 10 000 cycles there would only be about 10^{-6} stalls per committed instruction, while a block lock-time of 1000 cycles would yield about 10^{-7} stalls per committed instruction. A coarse-grained estimate of the performance penalty is to simply multiply the number of lock stalls with the lock stall-time. Thus, for the worst-behaving benchmark (*equake*) choosing 1000 cycles as the lock time would imply that for each instruction, the processor would stall $7*10^{-6}*1000 = 0.007$ cycles. The difference between the number of sub-page lock-stalls and page lock-stalls is not as large as the difference between sub-page overflows and page overflows however. For example, the difference between sub-page lock-stalls and page lock-stalls is about a factor of two for *SAP* meaning that using sub-pages can reduce the performance impact but not as much as it reduces the consumed relocation bandwidth.

We also studied the number of lock stalls for systems using hystereses. The results varied greatly across the various benchmarks and stall times. For most of the benchmarks the number of block lock-stalls was reduced by slightly less than a factor of two, but for *vpr* and *twolf* it was reduced between 5 and 10 times. The same holds for *mcf* if the stall time was 10 000 cycles. The effects of hystereses on sub-page and page lock stalls also varied across the different benchmarks. For some benchmarks it was extremely efficient, *e.g.*, SpecJBB where the number of page lock-stalls was virtually eliminated if hystereses was used. While hystereses does not help for all benchmarks, the great reduction in lock stalls for some of the benchmarks justifies its use given the small losses in compressibility as discussed in Section 5.1.

5.3. Detailed performance results

Now after having built intuition into how performance is affected by various parameters we present the final cycle-accurate performance results in Figure 8. The results are shown as normalized IPC, but the baseline IPC numbers are shown above the bars. Note that the range of the y-axis does not include zero since the variations compared with the baseline are extremely small.

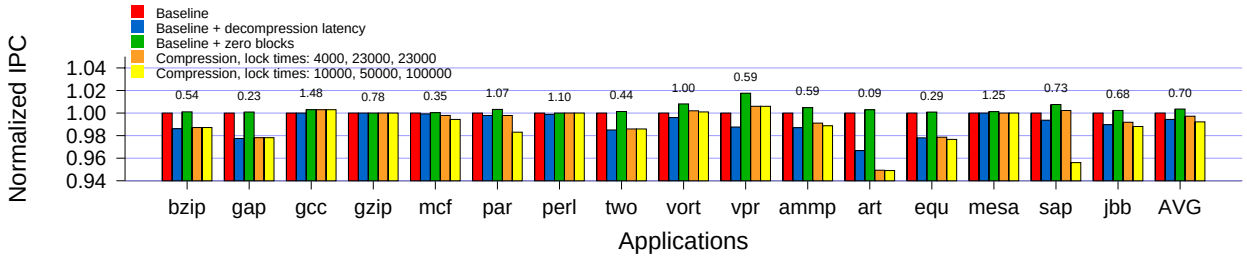

FIGURE 8. Normalized IPC results. The y-axis range does not include zero.

The first bar represents the baseline system. The second bar shows the effect of the increased memory-access latency due to decompression. On average, this reduces performance by 0.5% but for *art* the reduction is about *3%*. The third bar shows a baseline system but with the addition that zeroed blocks do not have to be loaded from memory. On average, this increases performance by slightly less than 0.5%. The fourth and fifth bars represent systems where all performance effects of the proposed compression scheme are factored in. The lockout times for the fourth bar are 4 000, 23 000, and 23 000, and for the fifth system they are 10 000, 50 000, and 100 000. The average performance degradation for the realistic latencies is less than 0.5% and for the worst-behaving benchmark (*art*) it is 5%. Using even longer latencies reduces average performance by less than 1% but now also SAP starts having problems with a degradation of about 4.5%. We can conclude that the performance impact of the proposed memory compression scheme is negligible.

6. Related work

Most of the earlier work on applying lossless compression to main-memory level data has considered dictionary-based methods that are more effective on bigger chunks of data. It was therefore natural to apply it to only memory pages that were selected as victims to be paged out to reduce performance losses of locating and decompressing pages. Douglis's [6] work aimed at reducing the paging overhead by freeing up memory in this fashion. Kjelso *et al.*[12] and Wilson *et al.* [21] demonstrated that a substantial reduction of paging overhead could be achieved by more accurately deciding which pages to compress. Wilson *et al.* [21] extended the virtual memory manager with a cost/benefit analysis to predict the reduction in page fault rate given an assumed compression ratio and use it to adaptively size the compressed portion of the memory. More recently, De Castro *et al.* [4] improved upon that by reducing the paging overhead further.

Despite the fairly significant decompression latencies, these systems could in many cases demonstrate performance gains because the lower paging overheads dwarfed the performance losses due to less effective access to compressed portions. In the IBM MXT technology [1][19][20], all memory contents are compressed. As noted earlier, reported decompression latencies are in the ball park of a memory access [20]. Moreover, to locate compressed data involves an indirection through a memory-resident table [19]. To shield the processors from these devastating latencies, a large (32 MB), tertiary cache sits in front of the compressed memory. Our approach is to practically eliminate the latency overhead altogether by using a compression technique with a negligible decompression latency and an effective translation mechanism between the linear and compressed address spaces. Thus, our method does not rely on huge tertiary caches.

As already mentioned, our work is inspired by the frequent-value locality property first discovered by Zhang *et al.,* [23] and refined by Yang and Gupta [22]. Alameldeen and Wood [2][3] exploited this finding in their frequent-pattern compression algorithm applied to caches and we have investigated in-depth how effective zero-aware compression algorithms are at in-memory data. A property related to frequent-value locality is that different data can be partially the same. For instance, consecutive addresses only differ in one position. This property has been exploited to compress addresses and data on memory-interconnects in [5][7].

7. Conclusion

In this paper a novel main-memory compression scheme has been proposed and evaluated. A key goal has been to practically remove decompression and translation overhead from the critical memory access path so that a balanced baseline system with little or no paging would not suffer from any performance losses. The compression scheme addresses this by using a fast and simple compression algorithm previously proposed for cache compression. It exploits our observation that not only memory words, but also bytes, and entire blocks and pages frequently contain the value zero. In fact, we have shown that simple zero-aware compression algorithms are efficient when applied to in-memory data since as much as 30% of all cache blocks and 55% of all bytes in memory are all zero.

Further, the compression scheme uses a memory layout that permits a small and fast TLB-like fast structure to locate the compressed blocks in main memory without a memory indirection. Finally, it is shown that by arranging a memory page logically into a hierarchical structure with a small slack at each level, and by using a DMA-engine to move data when the compressibility of a block is changed, it is possible to keep the performance overhead to a minimum while at the same time keeping fragmentation at a low level. We have shown that the amount of main memory in a machine can be reduced by 30% with a performance degradation of 0.2% on average.

As a final remark, we have deliberately focused on eliminating performance losses of existing compression schemes. However, compression can also potentially increase performance by reducing bandwidth requirements, and reduce power dissipation. Thus, we believe that this technology is now getting mature for adoption.

Acknowledgments

This research has been supported by grants from the Swedish Research Council and the Swedish Foundation for Strategic Research under the ARTES program. Equipment grants from Sun Microsystems have enabled computer simulations needed for this work.

References

[1] Abali B, Franke H, Shen X, Poff D and Smith B. Performance of Hardware Compressed Main Memory. *Proceedings of 7th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 73-81, January 2001.

[2] Alameldeen A and Wood D. Adaptive Cache Compression for High-Performance Processors. *International Symposium on Computer Architecture (ISCA-31)*, pages 212-223, June 2004.

[3] Alameldeen A. R. and Wood D. A. *Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches*. Technical Report 1500, Computer Sciences Department, University of Wisconsin-Madison, April 2004.

[4] De Castro R. S., Do Lago A. P. and da Silva D. Adaptive Compressed Caching: Design and Implementation. In *Proceedings of the 15th Symposium on Computer Architecture and High-Performance Computing*, pages 10-18, November 2003.

[5] Citron D. and Rudolph L. Creating a Wider Bus Using Caching Techniques. *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture*, pages 90-99, February 1995.

[6] Douglis F. The Compression Cache: Using On-line Compression to Extend Physical memory. *Proceedings of 1993 Winter USENIX Conference*, pages 519-529, January 1993.

[7] Farrens M. and Park A. Dynamic Base Register Caching: A Technique for Reducing Address Bus Width. *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 128-137, May 1991.

[8] Hallnor E.G. and Reinhardt S. K. A Unified Compressed Memory Hierarchy. *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 201-212, February 2005.

[9] Huffman D. A Method for the Construction of Minimum Redundancy Codes. *Proceedings of the IRE*. Vol. 40, No. 9, pages 1098-1101, 1952.

[10] Kaplan S. Compressed Caching and Modern Virtual Memory Simulation. *Ph.D. thesis. The University of Texas at Austin*, January 1999.

[11] Kjelso M., Gooch M. and Jones S. Design and Performance of a Main Memory Hardware Data Compressor. In *Proceedings of the 22nd Euromicro Conference*, pages 423-429, 1996.

[12] Kjelso M., Gooch M. and Jones S. Performance Evaluation of Computer Architectures with Main Memory Data Compression. *Journal of Systems Architecture*. Vol. 45, pages 571-590, 1999.

[13] Magnusson P S, Dahlgren F, Grahn H, Karlsson M, Larsson F, Lundholm F, Moestedt A, Nilsson J, Stenstrom P and Werner B. SimICS/sun4m: A Virtual Workstation. *Proceedings of Usenix Annual Technical Conference*, pages 119-130, June 1998.

[14] Núnes J and Jones S. Gbit/s Lossless Data Compression Hardware. *IEEE Transactions on Very Large Scale Integration (VLSI) systems*. Vol. 11, No. 3, June 2003.

[15] Salomon D. Data Compression. The Complete Reference. *Springer-Verlag*, New York, pages 151-218, 2000.

[16] Sherwood T, Perelman E, Hamerly G and Calder B. Automatically Characterizing Large Scale Program Behavior. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45-57, October 2002.

[17] Storer J. A., and Szymanski T. G. Data Compression via Textual Substitution, *Journal of the ACM*. Vol. 29, pages 928-951, 1982.

[18] Teller P. Translation-Lookaside Buffer Consistency. *IEEE Computer*, Vol. 23, No. 6, pages 26-38, June 1990.

[19] Tremaine R. B., Franaszek P. A. , Robinson J. T. , Schulz C. O., Smith T. B., Wazlowski M. E., Bland P.M. IBM Memory Expansion Technology (MXT). *IBM J. of Research & Development.* Vol. 45, No. 2., pages 271-285, March 2001.

[20] Tremaine R. B., Smith T. B., Wazlowski M., Har D., Mak K-K, Arramreddy S. Pinnacle: IBM MXT in a Memory Controller Chip. IEEE Micro, Volume 21, Issue 2, pages 56-68, March 2001.

[21] Wilson P., Kaplan S., Smaragdakis Y. The Case for Compressed Caching in Virtual Memory Systems. In *Proceedings of the USENIX Ann. Technical Conference*, pages 101-116, June 1999.

[22] Yang Y. and Gupta R. Frequent-Value Locality and its Applications. In *ACM Trans. on Embedded Computing Systems*, Vol. 1, pages 79-105, November 2002

[23] Zhang Y., Yang J. and Gupta R. Frequent Value Locality and Value-centric Data Cache Design. *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 150-159, November 2000.

[24] Ziv J., and Lempel A. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory.* Vol. 23, No. 3, pages 337-343, 1977.