RC22107 (W0106-018) June 13, 2001
Computer Science

IBM Research Report

Performance of Memory Expansion Technology (MXT)

Dan E. Poff, Mohammad Banikazemi, Robert Saccone,
Hubertus Franke, Bulent Abali, T. Basil Smith
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Performance of Memory Expansion Technology (MXT)

Dan E. Poff, Mohammad Banikazemi, Robert Saccone,
Hubertus Franke, Bulent Abali, T. Basil Smith

IBM T.J.Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598
{poff,mb,rsaccone,frankeh,abali,tbsmith}@us.ibm.com

Abstract

A novel memory subsystem called Memory Expansion Technology (MXT) has been built for fast hardware compression of main memory contents. This allows a memory expansion to present a “real” memory larger than the physically available memory. This paper provides an overview of the memory compression architecture, the OS support and an analysis of the performance impact of memory compression while running multiple benchmarks. Results show that the hardware compression of main memory has a negligible penalty compared to an uncompressed memory, and for memory starved applications it increases performance significantly. We also show that applications’ memory contents can be compressed usually by a factor of two.

1. Introduction

Data compression techniques are extensively used in computer systems to save storage space or bandwidth. Both hardware and software based compression techniques are used for storing data on magnetic media or for transmission over network links. While compression techniques are prevalent in various forms, hardware compression of main memory contents has not been used to date due to its complexity. The primary motivator for a compressed main memory system is savings in memory cost and space savings for tightly packed systems, such as for 1U (1.75”) thin, rack-mounted systems. Compression increases the amount of memory, or in cost sensitive applications it provides the expected amount of memory at a smaller cost. Recent advances in parallel compression-decompression algorithms coupled with improvements in the silicon density and speed now makes main memory compression practical [1,2,3,4].

A high-end, Pentium-based server class system with hardware-compressed main memory, called the Memory Expansion Technology (MXT), has been designed and built [3]. In this paper, we present a brief overview of the hardware and software technology required to enable the MXT technology, and provide detailed performance results and main memory compressibility of several benchmarks. Results show that two-to-one compression (2:1) is practical for most applications and that the performance impact of compression is insignificant, and for memory starved applications main memory compression improves performance significantly. Two-to-one compression effectively doubles the amount of memory; or in cost sensitive applications it provides the expected amount of memory for ½ the expected cost or even less. Larger memory configurations require more expensive, higher density memory modules due to the four memory-slot limit of a typical system. Therefore, an additional cost benefit of MXT is being able to use less expensive, lower density modules.

Observations show that main memory contents of most systems, operating system and applications included, are compressible. Relatively few applications’ data that are already compressed or encrypted, cannot be further compressed. In the MXT system, a Compressed Memory/L3 cache controller chip is central to the operation of the compressed main memory [3]. The MXT architecture adds a level to the conventional memory hierarchy. Real addresses are the conventional memory addresses seen on the processor external bus. Physical addresses are the addresses used behind the controller chip for addressing the compressed memory, also referred to as physical memory in this paper. The controller performs the real to physical address translation and compression /decompression of L3 cache lines. The processors are off-the-shelf Intel processors. They run with no changes in the processor or bus architecture. Standard operating systems such as Windows NT, Windows 2000 and Linux, run on the new architecture with no changes for the most part. However, a corner case exists, in which physical memory may be exhausted due to incompressible data. Standard operating systems are unaware of this problem. Hence, software support is needed to prevent the physical memory exhaustion. The amount of physical memory required changes with the compressibility of the memory contents. For example, a program starting with zero-filled buffers will require more physical memory as the buffers are loaded from disk. Hence, the physical memory requirements change as the program runs, requiring constant monitoring, and tuning as well as a recovery process if memory usage approaches the limit of the physical size. This corner case and the compressed memory management are handled by small modifications in the Linux kernel and by a set of user level services and a device driver in the Windows NT and Windows 2000 operating systems.

The main contributions of this paper are as follows. We provide an overview of the memory compression hardware and software support. Combined software/hardware design allows applications to run and take advantage of compression transparently. Using benchmarks we show the cost/performance benefits of doubling the memory size due to compression. We further show that a number of applications' main memory contents can be compressed effectively.

In the following section we give an overview of the MXT hardware. In Section 3 we describe the memory compression support added to the Linux and the Windows operating system. In Section 4, we present experimental results of various industry benchmarks on the MXT system and we examine the compressibility of various applications' memory contents. Conclusions are in Section 5.

2. Overview of MXT Hardware

In an MXT system the physical memory (SDRAM) contains compressed data and can be up to 16 GB in size. A third level cache (L3) is introduced. The L3 cache is a shared, 32-MB, 4-way set associative write-back cache with 1-KB line size and is made of double data rate (DDR) SDRAM. The L3 cache contains uncompressed cached lines and hides the latency of accessing the compressed memory. The L3 Cache/Compressed Memory Controller (Champion North Bridge CNB 3.0 HE component of the Pinnacle server chipset developed in cooperation with Serverworks) is central to the operation of the MXT system. The L3 cache appears as the main memory to the processors and I/O devices, and its operation is transparent to them. The controller compresses 1 KB cache lines before writing them into the physical memory.

The compression algorithm is a parallelized generalization [2] of the Lempel-Ziv algorithm known as LZ1. The compression scheme stores compressed cache lines to the physical memory in a variable length format. The unit of storage in physical memory is a 256-byte *sector*. Depending on its compressibility, a 1 KB cache line may occupy 0 to 4 sectors in the physical memory. Due to this variable length format, the controller must translate real addresses to physical addresses. A 1 KB cache line (real) address is mapped to 0 to 4 sector (physical) addresses in the physical memory. The real address is the conventional address seen on the processor chip's external bus. The physical address is used for addressing the sectors in the compressed physical memory. The memory controller performs real to physical address translation by a lookup in the Compression Translation Table (CTT), which is kept (uncompressed) at a reserved location in the physical memory. The CTT size is 1/64th of the real memory size.

Each 1 KB cache line address maps to one entry in the CTT, and each CTT entry is 16 bytes long (therefore, the 64 to 1 ratio between and the real memory size and the CTT size.) A CTT entry contains control flags and four physical addresses each pointing to a 256-byte sector in the physical memory. Different physical memory occupancies result from compressing 1 KB cache lines with different compression characteristics. For example, a 1 KB cache line, which does not compress occupies 4 sectors, i.e. 1 KB of physical memory. A cache line that compresses by 2:1, will occupy only two sectors in the physical memory (512 bytes) and the CTT entry will contain two addresses pointing to those sectors. The remaining two pointers will be null. For a cache line that compresses to less than 120 bits, for example a cache line full of zeros, a special CTT format called *trivial line* format exists. In this format, the compressed data are stored entirely in the CTT entry replacing the four address pointers. Therefore, a trivial line of 1 KB occupies only 16 bytes in the physical memory resulting in a compression ratio of 64:1. Another memory saving optimization implemented in the controller is sharing of sectors by *cohort cache lines.* If two 1 KB cache lines are in the same 4 KB page, they are called *cohorts.* Two cohorts may share a sector provided that space exists in the sector. For example, two cohorts each compressing to 100 bytes may split and share a sector since their total size is less than the sector size of 256 bytes. The compression operations described so far are entirely done in hardware with no software intervention.

Note that the selection of the 1 KB line size was influenced by many factors. Directory size, which grows inversely proportional to the cache line size for a given cache size, and the compression block size that effects the compression efficiency were the two most significant factors for the 1 KB line size [3]. Shorter lines may not compress well and longer lines may impact performance due to longer compress/decompress times. In this implementation, multiple compressors are used for performance reasons, but the dictionary is shared jointly to achieve a better compression ratio. Another technique that is employed to decrease memory-access latency is to provide the critical 32 bytes of data to the processor bus as soon as it is decompressed, rather than waiting until the entire 1 KB line completed. This technique, on average, reduces the decompression latency to one-half.

In addition to the above operation the compressed memory controller provides fast page operations, such as page moving and page zeroing, which perform significantly faster than if issued through regular memory operations. Fast page operations work on 4-KB pages, same as in the x86 architecture. The speed increase is achieved merely by updating pointers in the CTT entries, rather than moving bulk data with the processor.

The decompression latency is brought down significantly through the usage of parallel compression techniques and the utilization of a deep memory hierarchy. Memory hierarchy employing multiple cache levels have been used for many years to reduce the effect of main memory access times, particularly as the disparity has grown in the past decade between processor bus speeds and memory bus speeds. In the MXT architecture the additional L3 cache captures many accesses that would go to main memory for miss retrieval. The sizes of L2 to L3 are 256 KB and 32 MB, respectively, leading to a low global miss rate. In addition the L3 cache is shared, four-way set associative and write-back. These characteristics, along with the large line size of 1 KB in the L3 allows for a low miss rate to the main memory. Achieving a low miss rate requires the large 32MB L3 cache, however the L3 cache size is limited by the L3 directory size that can be supported on the controller.

In the MXT architecture, I/O data move through the shared L3 cache, unlike in the traditional L1/L2 cache organizations. It is appealing to have direct I/O access in to the compressed physical memory in terms saving disk space and I/O bandwidth due to the typically higher than 1.0 compression ratio. However, this would have been somewhat difficult and impractical since it would require additional compression circuitry in the I/O path and since data are possibly scattered in several non-contiguously located 256-byte sectors.

Another aspect of this architecture is the real-to-physical address translation performed by the MXT memory controller. The translation is performed transparently to the processors, I/O devices, and software. This has the advantage of being able to use stock CPUs and I/O peripherals and without any changes in the software (except for the memory management subsystem of the OS.) The translation is performed only for L3 cache misses which are in the low single digits due to the large L3 size. For current processor/memory organizations, combining real-to-physical address translation with virtual-to-real address translation appears neither useful nor practical, unless processors and memory controllers become integrated on a single chip in the future.

The additional cost for the MXT memory controller is estimated at ~$60-100 plus the cost for the L3 cache SDRAM. In return, for a 2GB system supporting 4GB of real memory, the system is approximately $2400 cheaper based on the memory prices effective at the time of writing this paper.

3. The MXT Memory Management Software

Compressed memory hardware allows an operating system to use a larger amount of real memory than physically exists. During the boot process, the hardware BIOS reports larger memory size than the installed physical memory. For example, in an MXT system we with 512 MB of installed SDRAM and the BIOS may reports having 1 GB of memory to the operating system. The main problem in such a system occurs when applications fill the memory with incompressible data, although more memory than physically available has been committed. In these situations, the common OS is unaware that the physical memory may be running out. For example, a 1024 MB system may have only 600 MB allocated and therefore may appear to have 424 MB free memory. However, due to low compressibility of the allocated memory, the physical memory usage may be near the 512 MB physical memory limit. Therefore, if the free memory is allocated, or if compressibility of the already allocated 600 MB further decreases, then the system will run out of physical memory, even though it appears to the OS that there is 424 MB free memory. Common operating systems do not distinguish between real and compressed physical memory nor do they deal with out-of-physical-memory conditions. The MXT memory management software addresses this problem. The general mechanisms underlying the MXT memory management software that prevent physical memory from running out can be described as follows:

A: Detect physical memory utilization
   1. Either by polling or through interrupts it detects physical memory utilization and exhaustion.
   2. Detect excessive I/O activity to adjust various thresholds to ensure forward progress [6].

B: Reclaim real memory and zero out freed pages to reduce utilization
   1. *Pageable pages*
      a) Make VMM believe that it is running out of memory and cause shrinking of file caches, and cause the paging daemon to move dirty pages to the swap disk. Pages freed are cleared with zeros, therefore physical utilization decreases, or
      b) Dispatch memory eater tasks/processes that allocate big chunks of memory stealing them from other processes. Then, clear the pages, while holding on to them as long as the physical utilization is high.
   2. *Non-pageable pages* (e.g. drivers and kernel extensions)
      a) Reserve an amount in physical memory equal to the non-pageable memory size.
      b) Force drivers to free memory (e.g. MXT aware drivers)

C: Steal CPU cycles to prevent further increase in utilization either by
   1. Descheduling processes, or
   2. Decreasing process priorities, or
   3. Activating a set of busy threads (one per CPU) to block processes from running.

In Linux, minor changes to the kernel were made in order to implement these mechanisms. For WinNT and Win2000, since kernel source code is generally not available, a combination of device driver and user-level services were implemented. The particular details of these implementations under Linux and Windows2000 are described in [15].

4. Performance Evaluation

The MXT memory system uses a relatively long 1-KB compression block size to be able to compress efficiently, since shorter blocks may not compress well. Due to the compression and decompression operations performed on these blocks in the memory controller, memory access times are longer than the usual. The 32-MB L3 cache contains uncompressed (1 KB) lines to reduce the effective access times by locally serving most of the main memory requests. Since this type of memory organization is relatively new, we present in the following a detailed performance analysis using the SPECint2000, the SPECWeb99 and a DB2 database regression test. In these experiments, we use MXT systems with dual 733 MHz Pentium III processors and a single disk drive. Dependent on the benchmark we run both Linux and Windows2000 and different memory sizes ranging from 512MB to 2GB physical memory. The MXT hardware was an early prototype, which had some of the performance enhancing features disabled such as bus defer response and processor IOQ depth limited to 1, due to hardware bugs. We compared the MXT hardware to a standard system with similar hardware characteristics except with no compression or L3 support.

4.1 SPECint2000 Results

The SPECint2000 benchmark suite is designed by the SPEC consortium to measure the performance of the CPU and the memory (http://www.spec.org/osg/cpu2000/) and requires approximately 256 MB of memory. There are 12 integer benchmarks in the suite. These are the *gzip* data compression utility, *vpr* circuit placement and routing, *gcc* compiler, *mcf* minimum cost network flow solver, *crafty* chess program, *parser* natural language processing, *eon* ray tracing, *perlbmk* perl utility, *gap* computational group theory, *vortex* object oriented database, *bzip2* data compression utility, and *twolf* place and route simulation benchmarks.

We ran the benchmarks three times, once on the standard system, and twice on the MXT system with the compression on and off. The difference between the standard and the MXT with compression off results give the performance impact of L3 cache. The difference between compression off and compression on results gives the performance impact of the compression-decompression hardware. The MXT system has a boot option that permits compression to be turned off. In that case, the system operates as a standard with an L3 cache and with non-compressed memory. Since compression/decompression hardware is disabled, the memory access latency is expected to be different than that of the compression-on case. We also recorded the number of L3 requests and L3 misses using the performance counters built into the memory controller chip.

Main results of the SPECint2000 experiments are shown in Fig. 1. The right most three columns are the average SPECint rates for the three systems that we considered. Due to strict reporting requirements of the SPEC consortium, we cannot publish the actual execution times. Therefore, we normalized the compression on and off results relative to the standard system results. Results show that on the average, MXT with compression-on is 1.3% faster than a standard system. The individual benchmarks *twolf*, *vpr, parser*, *gcc*, and *bzip2* perform 4.0 to 8.3% faster on the MXT system as shown in Fig.1. *Mcf* runs 1.1% faster on MXT. The L3 miss rates (Fig.2) and L3 request rates (Fig.3) for these six benchmarks reveal the reason: their miss rates are relatively small, but their L3 request rates are the highest among the twelve benchmarks. In other words, the working set of these six benchmarks fit in the 32 MB L3 cache and since they make large number of L3 requests, the L3 cache comprising double data rate (DDR) SDRAM gives a slight performance advantage over the standard system comprising regular SDRAM. On the contrary the benchmarks *vortex*, *gzip*, and *gap* respectively run 2.1%, 2.4%, and 10% slower on the MXT system than the standard system. Fig.2 shows that these three benchmarks have the highest miss rates among the twelve benchmarks (g*ap* that has the worst performance and the highest miss rate.) It also has the highest misses/second metric, which is 2.6 times greater than the next highest. Another observation is the relatively small L3 request rates for *crafty* and *eon*, which indicate that their working sets entirely fit in the L1 and L2 caches. Therefore, the L3 cache does not impact the performance of crafty and eon, which reveals itself as a negligible difference in their SPEC rates on Figure 1. Comparisons between MXT with compression and without compression show that the compression on case is 0.4% faster than the compression off case on the average. Figure 1 shows that *vpr* and *twolf* have the greatest difference in favor of the compression-on case. One possible explanation is that cache misses of these two benchmarks may result in large number of *trivial* line compression and decompressions. As explained before, a *trivial* line is an L3 cache line, which compresses to less than 120 bits and therefore is stored in a 16 byte CTT entry. Compression and decompression of trivial lines may have a smaller overhead than that of a line occupying a sector (256 bytes) or more in the physical memory. For example, cold cache misses at the beginning of execution will almost always result in trivial line compressions because the memory is filled with zeros initially. The compression off case runs faster than the compression on case for the *gap* and *vortex* benchmarks by 1.5 and 1.9% respectively. These have the first and third highest miss rates, and first and second highest misses per second.

In summary, the impact of the L3 cache and memory compression for the set of benchmarks is negligibly small considering that MXT doubles the amount of memory.

4.2. SpecWeb99 Results

In this test we measure the performance of webserving using the SpecWeb99 Benchmark comparing an MXT system with a standard system. The operating system is Linux and the webserver is TUX 2.0. The fileset of SPECweb99 was generated with the test option of wafgen.c enabled to make the files compressible. The standard SPECweb99 benchmark uses random character strings as the file content which is not realistic. Web content is generally compressible by a factor of 2 as shown later in Figure 9 and [14]. Web benchmarks including the SPECweb99 need to be changed to present a more realistic web content. Two system configurations with varying memory sizes and different number of Gigabit Ethernet adapters are analyzed. The left columns in Figure 4 show the achievable number of connections per second for the adapter (1EthGb) configurations. Since the SPEC consortium has strict rules for reporting results, we are not providing absolute numbers of achievable connections, but only relative numbers normalized with respect to the left hand most column (512MB MXT+1EthGB). Adding further memory to the system does not substantially increase the performance as the system is bandwidth limited. However, we observe that MXT can provide roughly the same performance with half the physical memory configured in the standard box. Adding an additional adapter in the 2EthGB configuration moves the bandwidth limitation from the 1GB memory size to the 2GB memory size. Again we observe that the same performance can be achieved under MXT with half the memory size of the standard box. We further note that at the same physical memory size the MXT systems delivers twice the performance. At this operating point, the I/O bandwidth is not the limiting factor. Instead, under MXT the OS is capable to keep a significantly higher number of webpages in the file cache (due to the compression) and thus does not have to fetch them as frequently from the disk. This is point is further supported in the section 5 about the application compressability.

4.3. DB2 Database Benchmark Results

The MXT system has been measured running an insurance company database schema. This configuration is primarily used within IBM as a quick regression test for ascertaining the impact of DBMS design changes (http://www.ibm.com /software/data/db2/). It is substantially less costly and quicker to run than complex benchmarks such as TPC-C, but is coarsely representative of the performance characteristics that might be expected. Several configurations were run on the prototype hardware: 512 MB with MXT off, 512 MB (1 GB expanded) with MXT turned on, 1 GB MXT off, 1 GB MXT on (2 GB expanded), and 2GB MXT on (4GB expanded) configurations. Benchmarks ran on the Windows 2000 operating system. Two runs were made for each configuration, a cold run where the file cache is initially empty, and immediately following that a second warm run, where the buffers have been "warmed" by the preceding cold run.

Figure 5 shows the performance benefits of MXT. For the 512 MB system when compression is on, it doubles the effective amount of memory and the benchmark runs 25% faster than the compression off case. For the 1 GB system when compression is on, it doubles the effective amount of memory to 2 GB and the benchmark runs 66% faster than the compression off case. It is interesting to note that the benefit of larger memory is more pronounced for this workload for larger memory sizes, and is indicative of both the smaller 512 MB memory and 1 GB memory configurations being memory starved. Finally, the 2GB configuration (4GB with compression on) contains the entire database in memory. The performance improvement in this case is 300%. Figure 5 also shows "performance twins" and "cost twins" to emphasize the benefits of MXT. Performance twins perform nearly identical, however the MXT-on twin costs less since its memory requirements are one-half. Cost twins have the same amount of physical memory, however the MXT-on twin performs better due to the doubling of the memory.

Figure 6 shows runtimes of the individual DB2 queries in a 4GB system after warm-up. The database is in memory at this point, so most I/O is eliminated. Generally, queries run a bit faster with compression on. Query 16 is an exception. This result is explained in Figures 7 and 8 which detail L3 cache accesses and misses for Queries 7 and 16. Query 7 has much higher L3 access and miss rates, and the compression ratio for this database is 2.68:1, resulting in improved bandwidth between the L3 cache and main memory with compression on.

However, the standard system generally runs faster. The 'standard system' used the same processors, twice the amount of SDRAM used in MXT, and a similar memory controller, except no compression or L3 support. On this early MXT prototype, performance-enhancing features of the processor bus was disabled due to hardware bugs, such as bus defer response and IOQ depth limited to 1, which is one possible explanation. Another possibility is higher L3 miss rates degrading overall performance compared to a standard system without an L3 cache.

4.3 Compressibility of Applications

Now that the performance of the MXT system is established, we turn our attention to the compressibility of main memory contents of various applications. We measured the compression ratios on the actual MXT hardware whenever possible. We used an estimation tool when MXT hardware was not available. Estimation tool samples the live memory contents while the application is running on a standard computer and predicts the compression ratio. Results show that most of these applications' main memory contents can be compressed usually by a factor of 2:1, justifying the real to physical memory ratio chosen for the MXT systems. The estimation tool is available for download at http://oss.software.ibm.com/developerworks/opensource/mxt/.

On the MXT hardware, the real and physical memory utilizations were recorded using an instrumentation register of the memory controller. The Sectors Used Register (SUR) reports to the operating system the amount of physical memory in use. A sampler program reads every two seconds the SUR register and the real memory utilization as reported by the OS and saves them in a file to be processed later. The measured memory values are for the entire memory. Therefore, in addition to the benchmark application's memory utilization, the measurements include possibly large data structures such as file cache and buffer cache that the OS maintains for efficient use of the system. In a post-processing step we took the average of the samples to produce the average compression ratio of a given benchmark.

Figure 9 shows the compression ratios for a few applications. Synopsis, Photoshop, MSDN Install and DB2 compression ratios were measured on the MXT hardware with the Windows2000 operating system. The Synopsis tool is used as a step in automating chip design. Photoshop compressibility varies significantly depending on the properties of the images being processed. For example, high-resolution topographical maps are incompressible, while images having less resolution, or areas of constant background compress well. Teiresias, from IBM Research, is an efficient algorithm for finding patterns in genetic structures. Teiresias ran on a stock PC and the compressibility was measured by an estimation tool that sampled the memory contents. This compressibility measurement was taken while analyzing the *E.coli* DNA. Microsoft Developer Network (MSDN) installation and most software installations compress poorly since the CDROM files are already compressed. The install program itself consumes only 4 MB. However, the associated file cache or NT standby pages fill the remainder of memory. The DB2 result is for an insurance company database schema. SPEC CPU 2000 is the average of the 12 integer benchmarks in the SPEC suite. Www.pc.ibm.com is a live web server used by IBM PC company customers. This result was obtained on a production web server and we used the estimation tool to sample memory contents.

Figure 10 shows the compressibility of the DB2 insurance database over time. The set of DB2 queries was run three times. The first run took 44 minutes, a cold run, reading the database from disk. The second and third runs took 12 minutes each, following 20 minutes idle time. Average compressibility was 2.68:1.

Overall the compressibility of applications, particularly those that require large memories (Webserving, databases), is at least 2:1. This justifies running the MXT system at a 2:1 real to physical ratio and one should not expect the physical exhaustion mechanism to kick in other than in "emergency" situations.

5. Related Work

Reference [2] considers an approach to compression that yields parallel speedup, while maintaining the compression efficiency of sequential approaches. Related work [5] focuses on the internal design of compressed random access memories. The issues of effective memory management in a compressed memory system are considered in [6]. A method for estimating the number of page frames as a function of physical memory utilization is described. They further model the residency of outstanding I/O as they transfer data into the memory when streamed through a cache, thus potentially forcing cache write backs that could increase the physical memory utilization. Using a time decay model they evaluate the system behavior using simulation. Reference [7] describes an approach to compression that removes the tight constraints of latency and bandwidth. This is accomplished by devising an architecture with two pseudo-levels – compressed and uncompressed memory, and the CPU operates only from the uncompressed region where the most frequently used pages are stored. Reference [8] introduces the concept of the compression cache, an intermediate level in the memory hierarchy that serves as a paging store. The author introduces this concept to take advantage of the improving speed of processors versus disk, and notes that the growing disparity between these system elements makes compression close to the processor an appealing feature. Reference [9] and the TinyRISC effort use compression to reduce embedded system code size. References [10] and [11] use compression techniques to increase branch prediction accuracy in microprocessors. Reference [12] describes algorithms and data structures for compressed memory machines. In contrast, our approach allows entire memory to be compressed unlike in [8], and it does not partition the main memory as compressed and uncompressed as in [7].

6. Conclusion

In this paper we described and evaluated a computer system with hardware main memory compression that effectively doubles the size of the main memory. We gave a brief overview of the MXT hardware and software technology. We measured the impact of compression on the application performance and determined that the hardware compression has a negligible penalty over an uncompressed hardware. We measured real and physical memory utilization of various applications and determined that main memory contents can be compressed usually by a factor of 2 or greater. Overall MXT provides an compelling argument to either increase performance through increased real memory size at the same price as standard systems or to reduce the cost of the system while retaining the same performance characteristics of standard systems.

References

[1] Hovis et al., “Compression architecture for system memory application,” US Patent 5812817, 1998.

[2] Franaszek, P, Robinson, J., Thomas, J. “Parallel Compression with cooperative dictionary construction,” In *Proc. DCC’96 Data Compression Conf.*, pp.200-209, IEEE 1996.

[3] Tremaine, R., Franaszek, P., Robinson, J., Schulz, C., Smith, T., Wazlowski, M., Bland, M.: “IBM Memory eXpansion Technology (MXT),” to appear in the IBM Journal of Research and Development Vol-2, 2001.

[4] Abali, B., and Franke, H.: “Operating System Support for Fast Hardware Compression of Main Memory”, *Memory Wall Workshop*, Vancouver, B.C., June 2000. Published as IBM Research Report No. RC21964, IBM, Yorktown Heights, NY 10598.

[5] Franaszek, P., Robinson, J., “Design and Analysis of Internal Organizations For Compressed Random Access Memory,” to appear in IBM Journal of Research and Development Vol-2, 2001.

[6] Franaszek, P., Heidelberger, P., Wazlowski, M.: “On Management of Free Space in Compressed Memory Systems”, *Proceedings of the ACM Sigmetrics*, 1999.

[7] Wilson, P, Kaplan, S., Smaragdakis, Y.: “The Case for Compressed Caching in Virtual Memory Systems”, *USENIX Annual Technical Conference*, 1999.

[8] Kjelso, M, Gooch, M., Jones, S.: “Empirical Study of Memory Data: Characteristics and Compressibility,” In *IEEE Proceedings of Comput. Digit. Tech*, Vol 45, No.1, pp 63-67, IEEE, 1998.

[9] Larin, Sergei Y.,et al: “Compiler-driven cached code compression schemes for embedded ILP processors,” *Proceedings of the Annual International Symposium on Microarchitecture*,1999.pp.82-92.

[10] Chen, I.-C.K., Coffey, J.T., Mudge, T.N.: Analysis of branch prediction via data compression, *Computer Architecture News* v. 24 Special Issue Oct 1996. pp. 128-137

[11] Kalamatianos, John, Kaeli, David R.: “Predicting indirect branches via data compression,” *Proceedings of the Annual International Symposium on Microarchitecture* 1998. pp. 272-281.

[12] Franaszek, P., Heidelberger, P., Poff, D.E., Robinson, J., “Algorithms and Data Structures for Compressed Memory Machines”, to appear in IBM Journal of Research and Development Vol-2, 2001.

[13] Vahalia, U: “Unix Internals, The New Frontiers”, Prentice Hall, ISBN 0-13-101908-2, 1996

[14] Abali, B., Franke, H., Poff, D.E., Shen, X., and Smith, T.B. "Performance of Hardware Compressed Main Memory", proceedings of The Seventh Int. Symposium High Performance Computer Architecture (HPCA-7), Monterrey, Mexico, Jan. 20-24, 2001., pp 73-81.

[15] Abali, B., Franke, H., Poff, D.E., Saccone, R., Schulz, C., Herger, L. and Smith, T.B. "Memory Expansion Technology (MXT): Software Support and Performance”, to appear in the IBM Journal of Research and Development Vol-2, 2001.

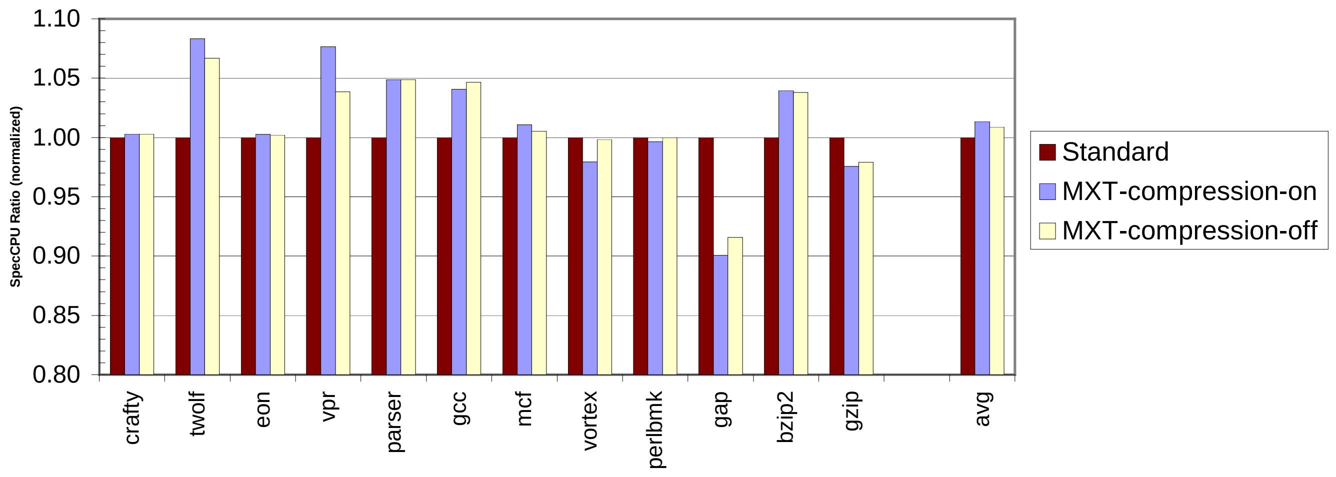

Figure 1: SPECint2000 results for three system configurations

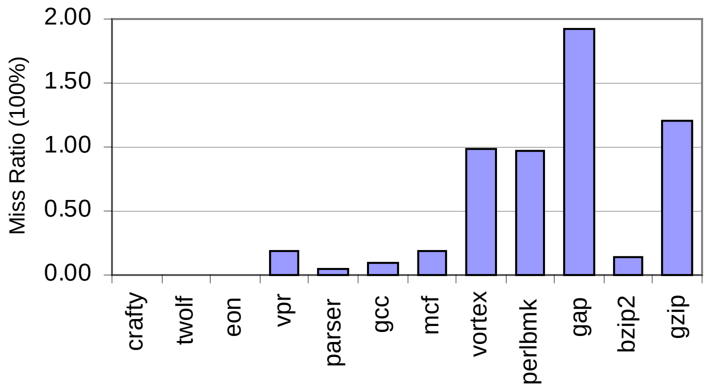

Figure 2: L3 Miss Ratio for SPECint2000 under MXT

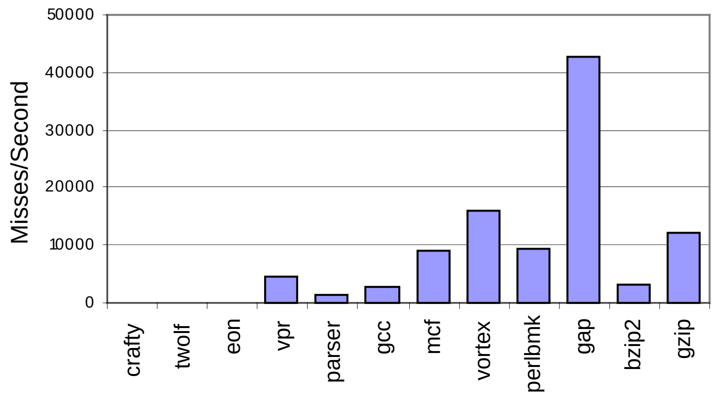

Figure 3: L3 request rates for SPECint2000

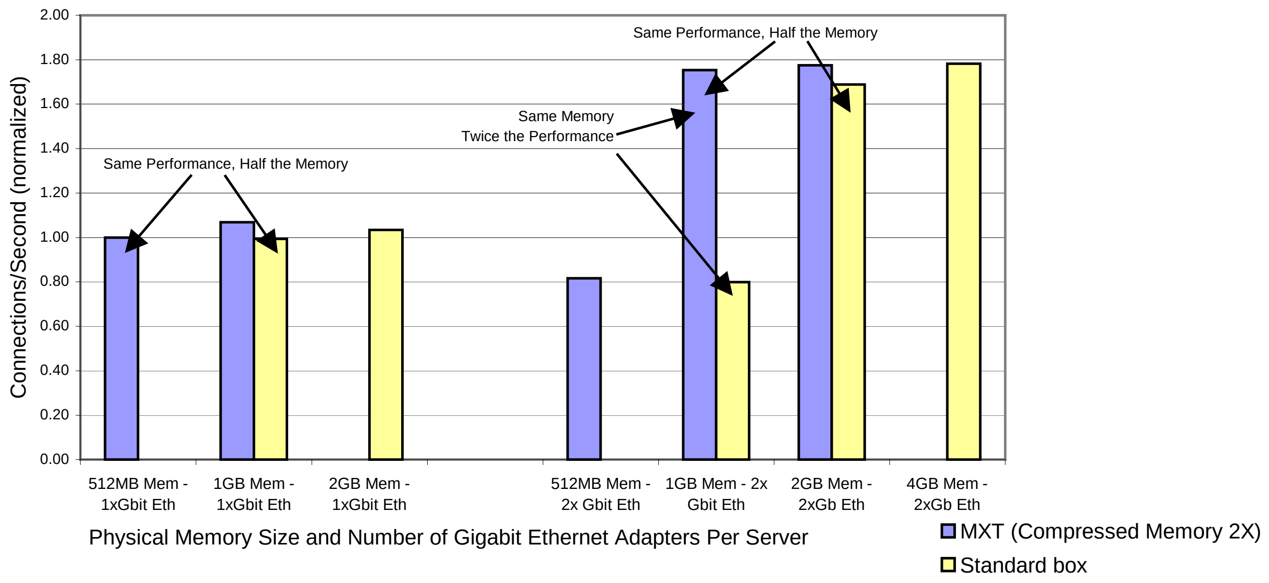

Figure 4: SpecWeb99 performance comparison

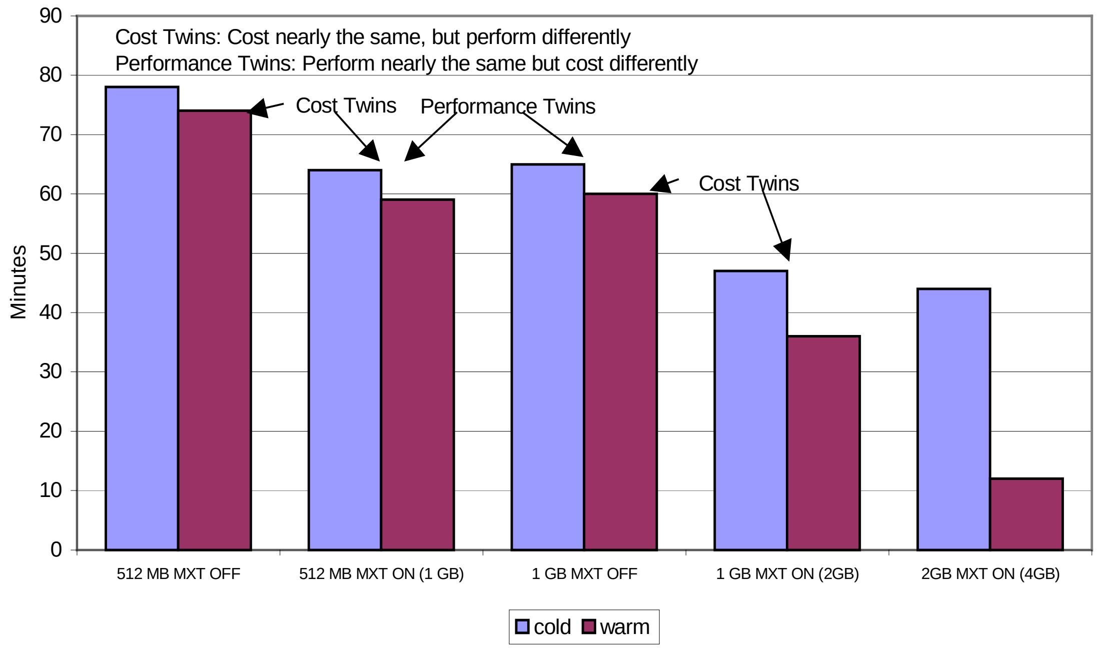

Figure 5: Database benchmark results for five different configurations

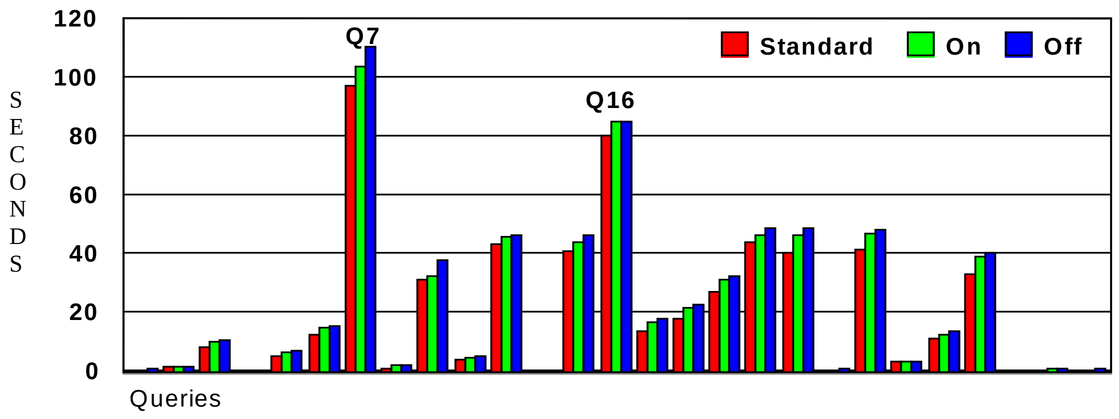

Figure 6: DB2 Query Runtimes

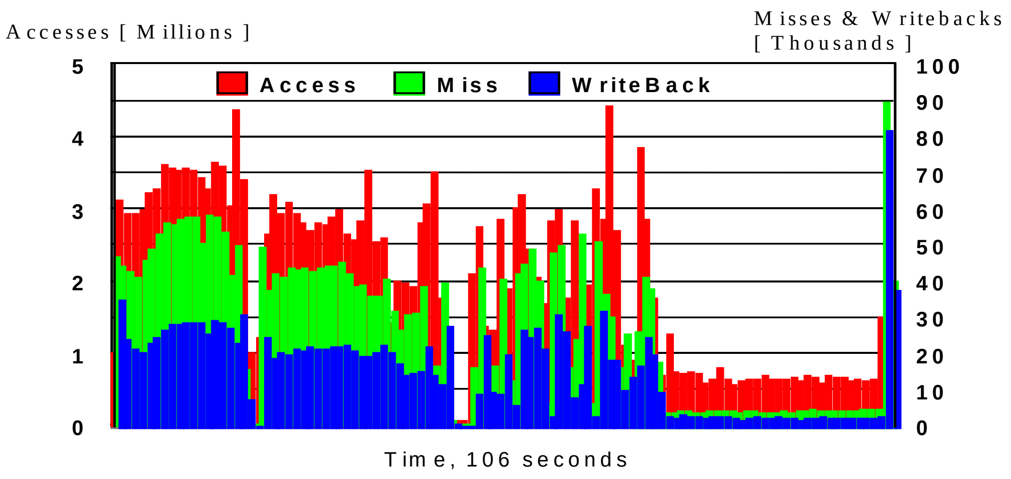

Figure 7: Query 7 L3 accesses vs. misses

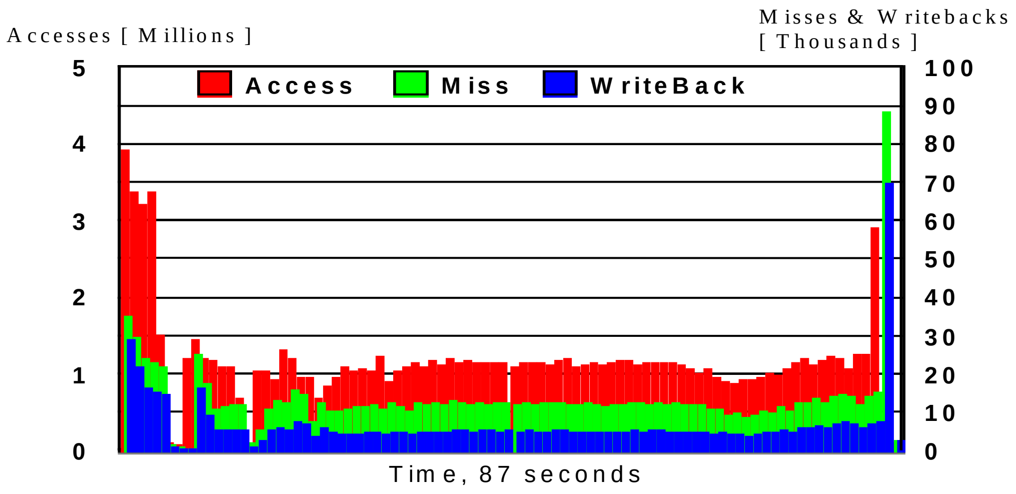

Figure 8: Query 16 L3 accesses vs. misses

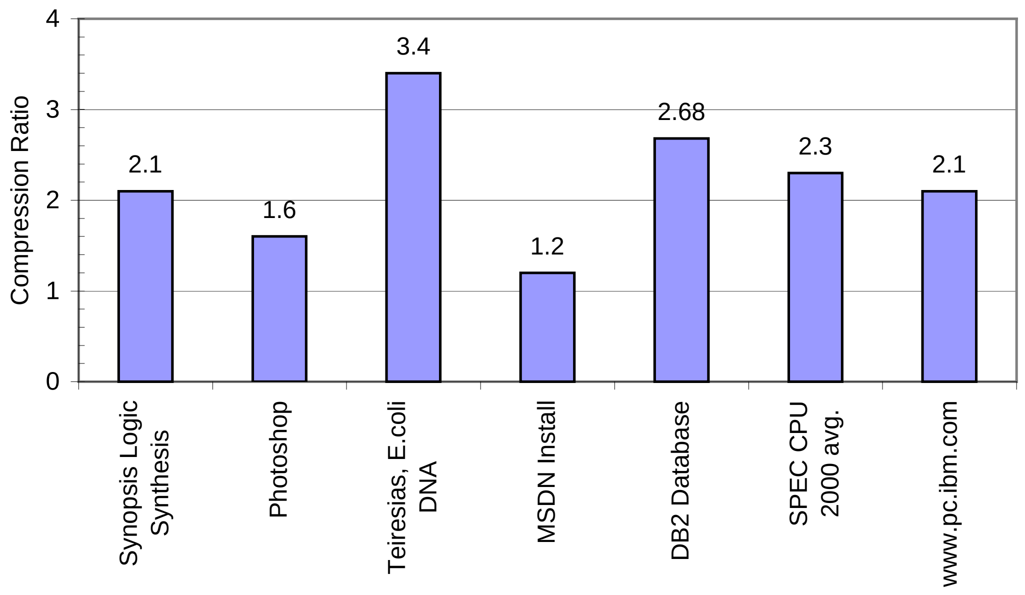

Figure 9: Compressibility of various applications

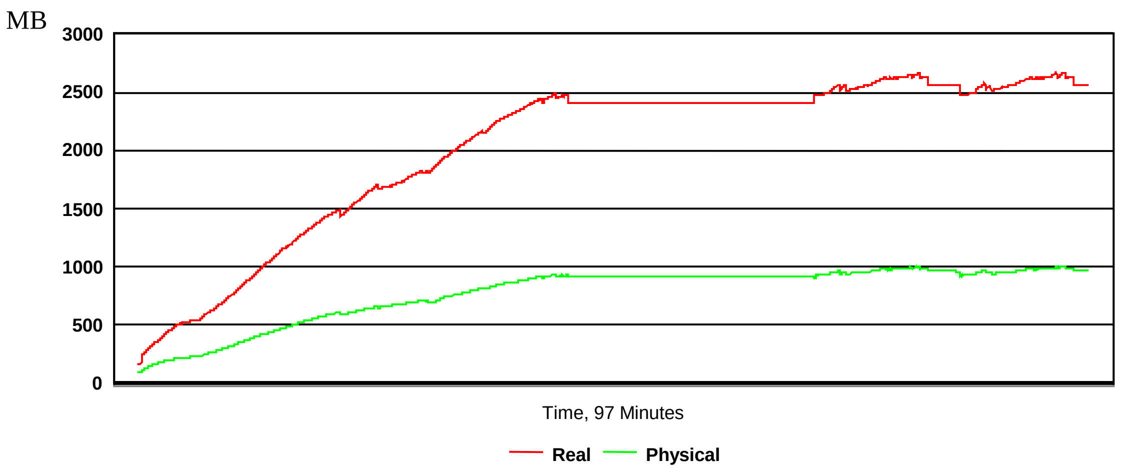

Figure 10: DB2 Database Compressibility: real vs. physical memory utilization