Memory Expansion Technology (MXT): Software support and performance

by B. Abali
H. Franke
D. E. Poff
R. A. Saccone, Jr.
C. O. Schulz
L. M. Herger
T. B. Smith

A novel memory subsystem called Memory Expansion Technology (MXT) has been built for fast hardware compression of main-memory content. 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, its OS support under Linux and Windows®, and an analysis of the performance impact of memory compression. Results show that the hardware compression of main memory has a negligible penalty compared to an uncompressed main memory, and for memory-starved applications it increases performance significantly. We also show that the memory content of an application can usually be compressed by a factor of 2.

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 content has not been used to date because of 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 (one height unit, or 1.75 in.) thin, rack-mounted systems. **Figure 1** illustrates the general cost and space-sharing benefits of memory compression. 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 silicon density and speed now make main-memory compression practical [1–4]. A high-end, Pentium**-based server-class system with hardware-compressed main memory, called Memory Expansion Technology (MXT), has been designed and built [3]. In this paper, we present an overview of the hardware and software technology required to enable MXT, and provide performance results and main-memory compressibility of a few benchmarks. Results show that two-to-one (2:1) compression is practical for most applications and that the performance impact of compression is insignificant; for

0018-8646/01/$5.00 © 2001 IBM

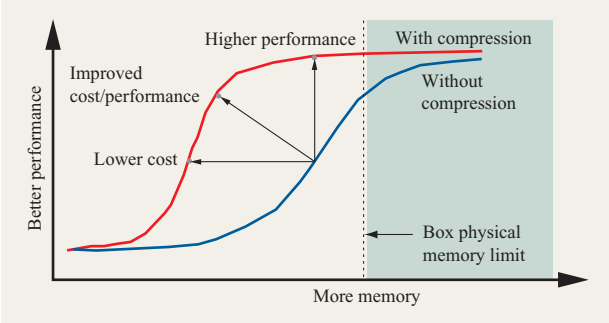

Figure 1

Motivation and benefits of main-memory compression; example performance vs. memory curves: with MXT (red curve); without MXT (blue curve).

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 half the expected cost or even less. The additional hardware cost of MXT over a standard platform is estimated to be less than $100 U.S. At the writing of this paper, memory prices were about $1 per megabyte for 128MB memory modules. Therefore, the break-even memory size for MXT was around 128 MB. Furthermore, the price per megabyte increases progressively with 512MB, 1GB, 2GB, and 4GB system memory configurations. Larger memory configurations require more expensive, higher-density memory modules because of the four-memory-slot limit of a typical system. Therefore, an additional cost benefit of MXT is its ability to use less expensive, lower-density modules.

Observations show that the main-memory content of most systems, operating system and applications included, is compressible. There are relatively few applications for which 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 external bus of the processor. 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 because of incompressible data. Standard operating systems are unaware of this problem. Hence, software support is needed to prevent physical memory exhaustion. The amount of physical memory required changes with the compressibility of the memory content. 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 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.

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. The authors further model the residency of outstanding I/O instructions as those instructions transfer data into the memory when streamed through a cache, thus potentially forcing cache writebacks that could increase physical memory utilization. Using a time-delay 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 pseudolevels—compressed and uncompressed memory. The CPU operates only from the uncompressed region, in which 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.

The primary contributions of this paper are as follows:

1. We describe the OS support software for managing the physical (compressed) memory. Our approach allows the entire memory to be compressed, in contrast to that of [8], and it does not partition the main memory into compressed and uncompressed segments as in [7]. Combined software/hardware design allows applications to run and take advantage of compression transparently.
2. Using benchmarks, we show the cost/performance benefits of doubling the memory size due to compression.
3. We further show that for a number of applications, main-memory content 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 Windows operating systems. In Section 4, we present experimental results of running a database benchmark on the MXT system and examine the compressibility of the memory contents of various applications. Conclusions are presented in Section 5.

2. Overview of MXT hardware

The organization of the MXT system is shown in **Figure 2**. The physical memory (SDRAM) contains compressed data and can be up to 16 GB in size. The third-level (L3) cache is a shared, 32MB, four-way set-associative writeback cache with 1KB line size. The L3 cache is composed 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 (the Champion Northbridge 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 1KB 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 1KB cache line may occupy 0 to 4 sectors in the physical memory. Because of this variable-length format, the controller must translate real addresses to physical addresses. A 1KB cache-line (real) address is mapped to sector (physical) addresses in the physical memory. The real address is the conventional address seen on the external bus of the processor chip. The physical address is used for addressing the sectors in the compressed physical memory. The

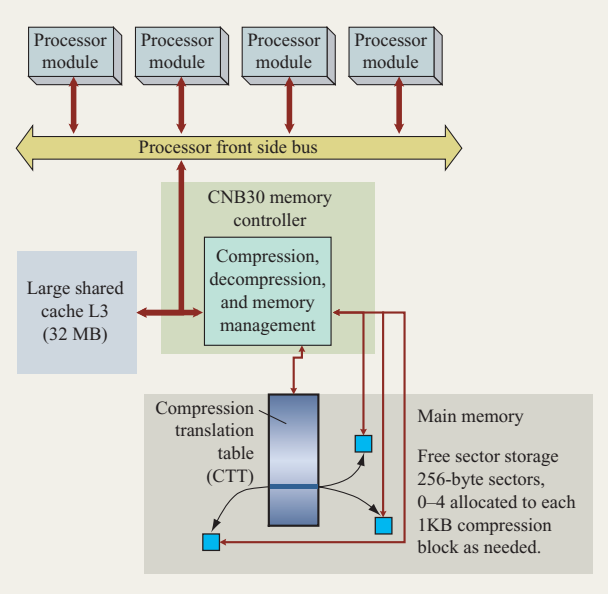

Figure 2

Overview of a system with memory expansion technology (MXT).

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 1/64 of the real memory size.

Each 1KB 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 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. **Figure 3** shows the different physical-memory occupancies that result from compressing 1KB cache lines with different compression characteristics. For example, a 1KB cache line which does not compress occupies four sectors, i.e., 1 KB of physical memory. A cache line that compresses by 2:1 occupies only two sectors in the physical memory (512 bytes), and the CTT entry contains two addresses pointing to those sectors, while the remaining two pointers are 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 1KB cache lines are in the same 4KB page, they are called *cohorts*. Two cohorts

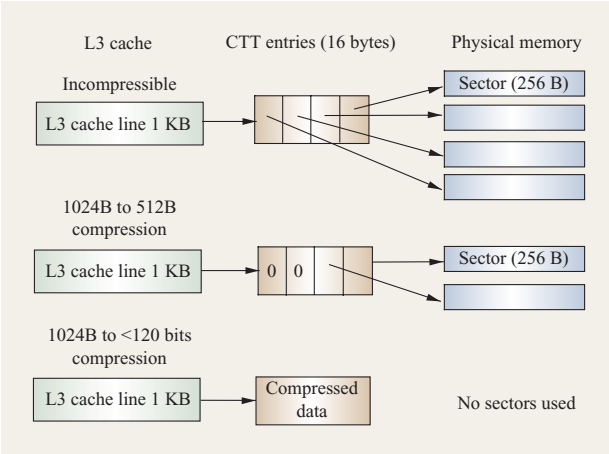

Figure 3

Physical memory occupancy compression of L3 cache lines through the CTT.

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 done entirely in hardware with no software intervention.

Note that the selection of the 1KB line size was influenced by many factors. Directory size, which grows in inverse proportion to the cache-line size for a given cache size, and compression-block size, which affects the compression efficiency, were the two most significant factors in selecting the 1KB line size [3]. Shorter lines may not compress well, and longer lines may affect performance because of longer compress/decompress times. In this implementation, multiple compressors are used for performance reasons, but the dictionary is shared 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 they are decompressed, rather than waiting until the entire 1KB line is completed. This technique, on average, reduces the decompression latency by one half.

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

The MXT architecture solves various problems that prevent successful deployment of main-memory compression: First, recent increases in the density of ASIC technology utilizing 0.18-μm packaging and smaller gate technologies, coupled with tools for custom design, allow the entire L3 cache/compressed-memory controller to be implemented on a single chip (CNB30). This chip includes the L3 directory but excludes the L3 data, which are in 32MB DDR SDRAM. The decompression latency is reduced significantly through the use of parallel compression techniques and a deep memory hierarchy. Memory hierarchies employing multiple cache levels have been used for many years to reduce the effect of main-memory access times, particularly since the disparity between processor-bus speeds and memory-bus speeds has grown in the past decade. In the MXT architecture, the additional L3 cache captures many accesses that would go to main memory for miss retrieval. The sizes of L2 and 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 writeback. These characteristics, along with the large line size of 1 KB in the L3, allow for a low miss rate to the main memory.

In the MXT architecture, I/O data move through the shared L3 cache, in contrast to traditional L1/L2 cache organizations. It would be appealing to have direct I/O access to the compressed physical memory in terms of 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 noncontiguously 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, and provides the advantage of being able to use stock CPUs and I/O peripherals 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 because of 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 are integrated on a single chip in the future.

3. MXT memory-management software

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. For Linux, minor changes to the kernel were made. For Windows NT and Windows 2000, since kernel source code is generally not available, a combination of device-driver and user-level services was implemented. In this section,

we describe a set of generic mechanisms for making any operating system compression-aware. We then follow with a description of compressed-memory management for Linux and Windows operating systems, which are designed and implemented by two separate teams of programmers. The differences between two implementations mostly have to do with the different constraints each OS imposes: For example, the Windows source code was not available, and compressed-memory management had to be done from outside the kernel, increasing the complexity of the code. However, both implementations use the same set of generic mechanisms described in this section.

The MXT hardware allows an operating system to use a larger amount of real memory than physically exists. During boot time, memory-size detection routines report to the OS twice the amount of physically available memory. For example, the particular MXT system we used has 512 MB of physical memory, but BIOS reports having twice that amount, 1 GB of memory. The main problem in such a system occurs when application(s) begin to fill the memory with incompressible data when the operating system commits more real memory than is physically available. In these situations, the common OS is unaware that the physical memory is being exhausted. Fortunately, minor changes in the virtual-memory management (VMM) of the OS kernel are sufficient to render the OS compression-aware.

At this stage we need to introduce and distinguish between the concepts of "static compression ratio" and "dynamic compression ratio." Static compression ratio is fixed in the BIOS setup. It is the ratio of the memory size BIOS reports to the operating system to the physical memory size, i.e., the actual amount of memory in the system. Static compression ratio is fixed at 2.0 in the current BIOS because measurements show that a 2:1 compression ratio is typical for a large number of applications. Dynamic compression ratio is a function of time and depends on the application. For example, during the initial phases of execution, an application will have most of its allocated memory filled with zeros and may therefore exhibit a compression ratio as high as 64:1. As execution progresses, the compression ratio will change depending on the memory content. It is possible to provide a static compression ratio of up to 64:1, but this implies that the OS must support 64 times the physical memory in terms of page frames. Since page-frame data structures are typically pinned, this would require a significant commitment of memory resources for an unlikely scenario.

Linux and Windows NT run without any changes in expanded (real) memory, as long as the compressed physical memory is not exhausted. The OS is shielded from the peculiarities of addressing the physical memory. However, to take full advantage of the memory-compression capability, the MXT system must eliminate the physical-memory exhaustion problem. The OS must be ready to reduce physical-memory utilization when it is near exhaustion by reducing real-memory usage or by increasing the dynamic compression ratio. The basic problem of running out of physical memory may be illustrated in the following example: A 1GB real-memory system may have only 600 MB allocated and therefore may appear to have 424 MB of free memory. However, because of the low compression ratio of the contents, the physical memory usage may be near the 512MB physical-memory limit. Therefore, if the remaining 424 MB of "free memory" is allocated, or if compressibility of the allocated memory decreases further, the system will run out of physical memory, even though it appears to the standard OS that free memory exists.

Compression-management software may deploy one or more of the following generic mechanisms to prevent physical memory exhaustion:

1. Detect physical memory utilization:
    a. Either by polling or through interrupts, it detects physical memory utilization and exhaustion.
    b. It then detects excessive I/O activity in order to adjust various thresholds to ensure forward progress [6].
2. Reclaim real memory and zero out freed pages to reduce utilization:
    a. Pageable pages:
        i. Make the 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, so that physical utilization decreases; or
        ii. Dispatch "memory-eater" tasks/processes that allocate large chunks of memory by stealing them from other processes. Then, clear the pages, while holding on to them as long as the physical utilization is high.
    b. Nonpageable pages (e.g., drivers and kernel extensions):
        i. Reserve an amount in physical memory equal to the nonpageable memory size.
        ii. Force drivers to free memory (e.g., MXT-aware drivers).
3. Steal CPU cycles to prevent further increase in utilization:
    a. Deschedule processes;
    b. Decrease process priorities; or
    c. Activate a set of busy threads (one per CPU) to block processes from running.

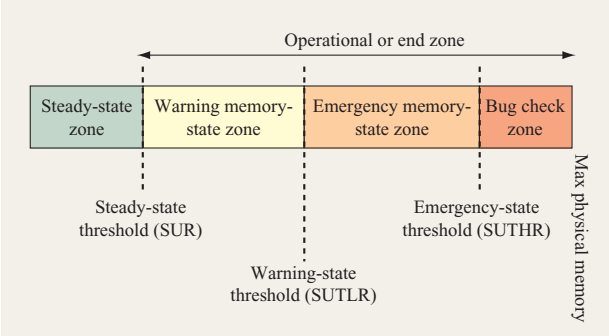

Figure 4

Physical memory utilization zones and operation mode of the MXT.

The compression management must be aware of physical-memory utilization. For this purpose, the MXT architecture exposes a set of monitoring registers. This register set is memory-mapped and accessible by the processors. The sectors-used register (SUR) reports the physical amount of memory utilized. The management software can periodically (e.g., every 10 ms or more) check the SUR to identify whether physical memory is near exhaustion. Alternatively, it can set two different threshold registers to raise an interrupt when the SUR value exceeds them: a) the sectors-used threshold-low (SUTRL) register raises an interrupt when the SUR exceeds SUTRL, and b) the sectors-used threshold-high (SUTRH) register raises a nonmaskable interrupt when SUR exceeds SUTRH.

A typical software implementation may have additional thresholds and zones of physical memory utilization (as shown in the example in **Figure 4**): steady-state, warning, emergency, and bug check zones. The last three zones are defined as the operational or end zone of the compressed-memory manager, in which it is actively working to reduce physical-memory utilization.

When physical-memory utilization reaches critical levels, further memory allocation must be prevented. To do so, we must make the OS believe that it is running out of real memory as well. This process is discussed in more detail further on; essentially, however, in Linux we raise the minimum number of free pages required by the kernel, while in Windows NT and Windows 2000 we aggressively allocate and hold the free pages in the system. If physical utilization does not return below a desired threshold, we continue to raise the free-memory threshold under Linux and continue to allocate pages under Windows. This in turn will force the OS to replenish the free-memory pool; hence, it will activate its swap daemon. Paging out pages by itself does not solve the problem, because neither OS clears freed pages. These pages must be cleared explicitly to reduce physical-memory utilization. When clearing pages, we utilize the fast page-zero operation, which has the effect of flushing a page out of the L3 cache (if present in L3), freeing any physical memory in use by the page, and writing the zero-bit pattern to the page's CTT entries.

Unfortunately, even this forced paging mechanism can be insufficient to reduce physical-memory utilization, because paging (to swap disks) occurs at the mechanical speed of disks. However, physical-memory utilization can increase at the speed of memory access. If physical-memory utilization continues to increase despite aggressive paging, processes must be slowed down—by decreasing their priority, by descheduling them, or by activating high-priority "busy threads" that steal cycles—until the paging mechanism has produced a sufficient decrease in physical-memory utilization.

In summary, the compression-management schemes described here serve to keep physical memory utilization below preset limits. Above those limits, the compressed memory system behaves the same as a conventional system with insufficient memory and therefore exhibits increased paging activity. In the following section, we describe our implementation for the Linux and the Windows NT and Windows 2000 (Windows NT/2000) operating systems.

Compression support under Linux

A compression-aware Linux kernel uses three primary mechanisms to control physical-memory utilization. First, the low-memory watermark of the free-page pool, which is a constant value in the conventional Linux kernel, is changed to a dynamically adjusted variable in the compression-aware kernel. When physical-memory utilization exceeds the preset thresholds, the low-memory watermark is raised, and the paging daemon is signaled to reclaim pages from the file-system cache and processes. This action will eventually increase the number of free pool pages. Second, when physical-memory utilization is above the preset thresholds, memory allocation is limited in order to prevent further exhaustion. As a result, the VMM is forced to replenish pages in the free-page pool. Third, reclaimed pages are zeroed before adding them to the free-page pool. An L3 cache line filled with zeros can be compressed to 1/64 of its original size, as discussed in Section 2. Therefore, zeroing freed pages helps reduce physical-memory exhaustion. In the following we describe these mechanisms in more detail.

In the Linux kernel, free pages are maintained within the free area by a "buddy" algorithm in order to ensure allocation of consecutive physical-memory ranges. The interface to this algorithm is the command pair *get_free_pages* (to allocate a power-of-two number of pages) and *free_pages* (to free a set of pages). On top of this page-frame management, Linux provides a *kmalloc*

interface for allocation of smaller chunks. It also provides an object *SLAB* allocator for its internal data structures [13].

The standard Linux 2.2 kernel by default tries to maintain a minimum of 256 free pages (1 MB) at all times for important kernel routines that should always succeed in page allocation, such as interrupt service routines. This low-memory watermark is a constant value named *freepages.min* and is set at boot time. During page allocation, if the number of free pages in the system (*nr_free_pages*) falls below freepages.min, the memory-management routines are called in sequence to trim SLAB caches, shrink memory-mapped files, get rid of shared-memory pages, and finally swap out process pages to the swap disk until the number of free pages increases above the freepages.min threshold. A Linux paging daemon, called *kswapd*, also exists that accomplishes the same objective in a lazy manner. The *kswapd* process "wakes up" once a second and tries to free up pages if nr_free_pages falls below another constant, *freepages.high*, which has been set to three times freepages.min at boot time.

In the compression-aware Linux kernel, the freepages.min and freepages.high watermarks are not constants, but are dynamically adjusted variables whose values are changed by the compression-management routines. In our prototype implementation, we check the physical memory utilization by polling the SUR register every ten milliseconds during the OS clock interrupts, and we recalculate new watermark values if needed. Polling adds a negligible overhead to the CPU utilization. When physical memory utilization is below the threshold th_acquire=0.85, the watermarks are kept at their original boot time values. When utilization is between th_acquire=0.85 and 1.00, watermarks are governed by the equation shown in **Figure 5**. The solid line indicates how much more physical memory usage is allowed. At every clock tick, watermarks are recalculated using the equation represented by the solid line. The Current=Target line (slope=1.0) and the objective function (solid line) intersect at th_danger=0.90, which means that physical-memory utilization may not exceed th_danger=0.90. When utilization is between th_acquire=0.85 and th_danger=0.90, the difference between target utilization (PTU) and current utilization (PMU) represents the amount of physical memory available for allocation. If more memory is demanded, the Linux virtual-memory manager must free up pages from elsewhere before handing them to the caller. (Note that physical utilization is also increased when processes change the memory content.) When utilization is above th_danger=0.90, PTU is smaller than PMU. Therefore, compression-management routines must reduce physical-memory utilization. This is accomplished by setting the free-page pool watermarks

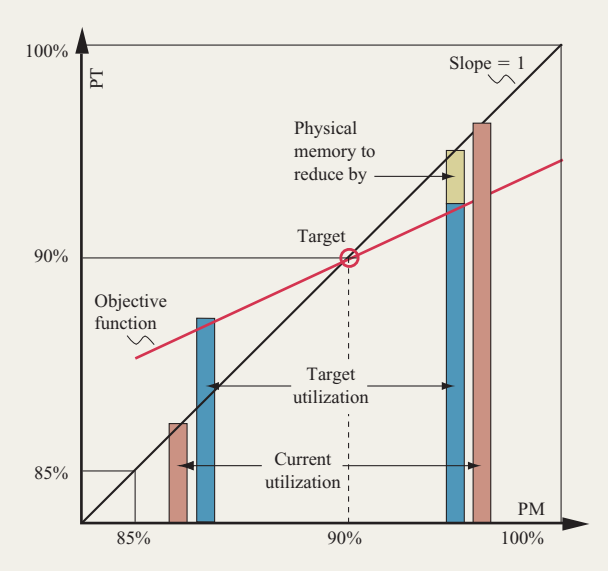

Figure 5

Memory manager objective function: Current (PM) vs. target (PT) physical utilization.

freepages.min and freepages.high to a number greater than the number of free pages. Then the kswapd daemon is signaled. Kswapd, by design, must begin freeing up pages until the number of free pool pages increases above the watermark freepages.high. Watermarks are calculated using the following equations. Let *UP* be the number of used pages in real memory and *FP* be the number of free pages in real memory (nr_free_pages), where the total number of real pages is $T = UP + FP$. Then, $UP \times PTU/PMU = MP$ gives the target real-page usage. Then freepages.high is set to $T - MP$. From Figure 5 it can be seen that if $PMU < 0.90$, $PTU > PMU$, which results in freepages.high $< FP$. If $PMU > 0.90$, then $PTU < PMU$, which results in freepages.high $> FP$. Thus, in the latter case the number of free pages is less than the required minimum, and kswapd must retrieve used pages and add them to the free-page pool.

The thresholds th_acquire and th_danger are empirically determined constants whose values are calculated at boot time as a function of L3 and physical memory size. Since L3 is a writeback cache, its contents and the corresponding physical memory locations are not coherent. In the worst case, the entire L3 may contain incompressible data, and the corresponding physical memory may be totally compressible; for example, consider an L3 filled with random data vs. the corresponding physical memory locations containing all zeros. In this worst-case scenario, an L3 cache flush into the physical memory will result in an increase in physical-

memory utilization by the L3 amount. To eliminate this hazard, we must keep in reserve free physical memory equal to the L3 amount. This means that the equation *th_danger* $< 1 -$ *L3 size*/*usable physical-memory size* must be satisfied, where *usable physical-memory size = installed physical-memory size − CTT size*. For a 512MB installed physical memory and a 32MB L3 cache system, the CTT size is 1024/64 = 16MB. Therefore, th_danger must be less than $1 - 32/(512 - 16) = 0.94$. An additional hazard may theoretically exist: When the threshold is exceeded, execution of kswapd and various compression-management routines may result in an additional increase in physical utilization, since they modify system structures in the memory, such as page tables. To cover this hazard, we reserve an additional, empirically chosen 4% of physical memory, therefore setting th_danger to 0.90 at boot time. Determining the guaranteed-to-be-safe reserve amount is a subject of ongoing studies.

Note that increasing the number of free pages in the system alone does not achieve the reduction in physical-memory utilization, because freed pages retain their content upon freeing in the conventional Linux kernel. The conventional Linux kernel zeros pages only at allocation time and only for nonkernel processes for security. In the compression-aware kernel, pages are cleared when they are freed; hence, all of the pages in the free-page pool are zero-filled. Recall that zero-filled pages consume 1/64 of the original size of their memory. Zeroing pages, coupled with dynamic adjustment of the free-page pool watermarks, allows us to control physical-memory utilization.

An application may rapidly change the content of its memory and therefore may rapidly increase the physical-memory utilization. In such cases, the compression-management schemes described so far may not be quick enough to reduce or keep memory utilization below th_danger=0.90 because the reduction in memory utilization is partly governed by the speed of swap-disk access, while the increase is governed by the speed of memory access. We use busy kernel threads to stall the execution of such applications. Kernel processes, one per processor, are created at boot time. These processes are normally idle. If physical-memory utilization increases above th_danger+2%=0.92, the idle processes are signaled to begin busy-spinning. This has the effect of stalling other processes in the system so that they cannot execute and increase physical-memory utilization until it falls below th_danger.

Compression support under Windows NT/2000

Under Windows NT/2000, an in-kernel solution requires source-code access, which was not available to us. Hence, we present here the approach called the "outside kernel solution," which consists of a device driver (Compmem on Windows NT 4, CMemW2k on Windows 2000), the compression management service (CMS), and one or more (depending upon memory size) "memory-eater" processes. The device driver is a kernel-mode component, while CMS and the memory eaters are user-mode components. The overall architecture of the compression-management software under Windows NT/2000 is shown in **Figure 6**. The functionality of each component is described in the following sections.

The device driver provides the following facilities:

1. Tracking of the physical utilization state.
2. Allowance for programming the thresholds of the low-physical-memory interrupts.
3. Broadcast notification of low-physical-memory interrupts to interested client applications.
4. Access to special page-operation functions of the compressed-memory controller.
5. Various memory-compression statistics.

In a manner similar to that of the Linux in-kernel implementation, the device driver categorizes memory utilization into three configurable states: steady, warning, and emergency. The driver provides client applications the ability to associate event-notification objects with each state. Upon detection of a memory-state change, the driver will reset the events associated with the prior memory state, and then set the events associated with the new memory state.

The device driver also supports user-mode access to MXT fast-page operations. The key page operation that the driver exposes to user-mode applications is the zero page operation. The application can pass down to the driver a virtual address in its process space and a length. The driver will convert the address from virtual to physical and invoke the zero page operation on each page in the range, thus reducing physical-memory utilization.

The compression management service (CMS) is the user-mode portion of the compressed-memory control system under Windows. It runs as a service process, which is equivalent to the role of UNIX** daemons under Windows. A service runs with real-time process priority so that it will preempt most other user-mode processes in the system. During initialization, the CMS determines the difference between real-memory size and physical-memory size. This result, called MaxMemToTakeAway, is the maximum amount of memory that must be removed from the virtual memory manager if an application(s) completely fills memory with incompressible data. Since the software is not part of the operating system, it is not possible to remove pages of real memory directly from the virtual memory manager page-frame database. Instead, memory is consumed by calling VirtualAlloc() to allocate pages. In Windows NT/2000 the maximum amount of

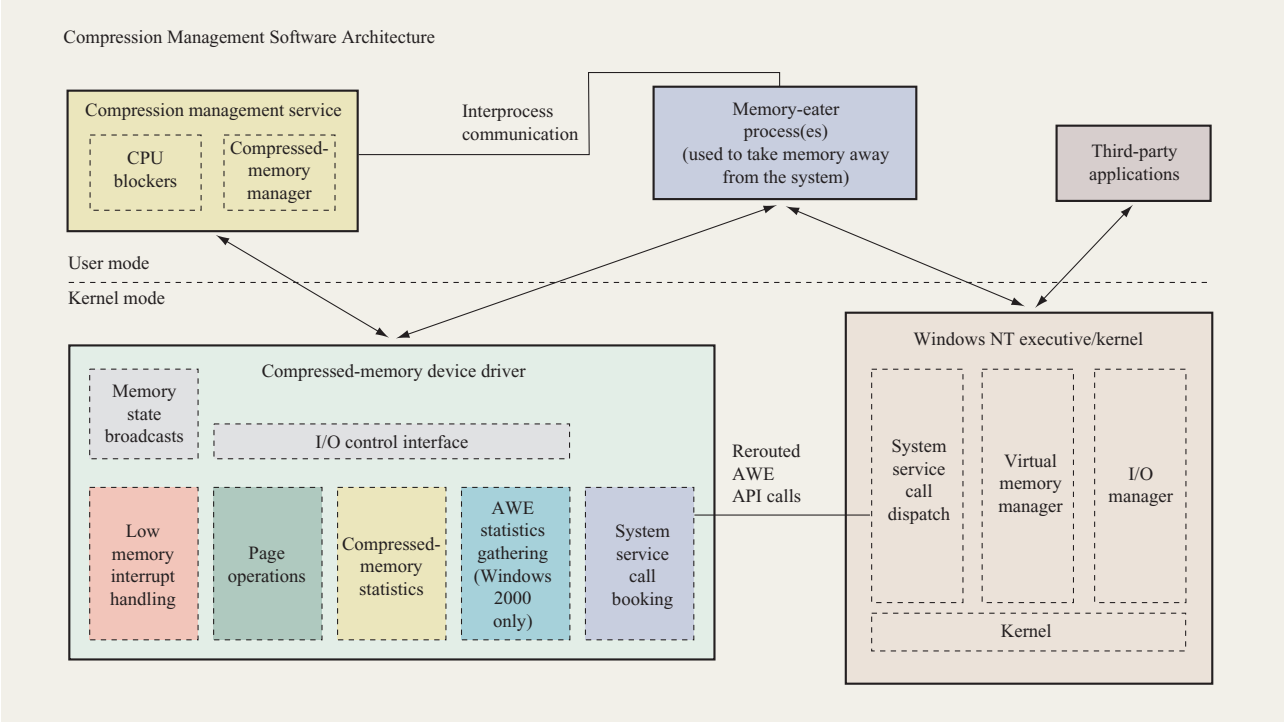

Figure 6

Compression management services under Windows.

memory that a process can allocate via VirtualAlloc() is between 2 and 3 GB, depending on the version of the operating system. This number includes code, static data, stack, and run-time overhead.

The CMS spawns one or more processes known as "memory eaters." Each memory eater is designed to allocate 1 GB of memory. Enough memory eaters are spawned so that in total they can allocate MaxMemToTakeAway bytes of memory. A block of shared memory is used as the means of communication between CMS and the memory eaters. The CMS directs the memory eaters to allocate or release memory being held by writing either a positive or negative adjustment value into the shared-memory block. The memory eaters update the adjustment as they perform their work, which is calculated as follows:

$$EndZoneSize = L3CacheSize + NonPagedPoolSize + (LockedPages \times PAGE\_SIZE) + CTTSize + SizeOfUncompressedRegions + BugCheckSize.$$

The BugCheckSize is the size of code that creates an abnormal end (abend) in the OS. If the emergency state threshold is breached, it is considered a fatal condition, and the OS is quickly brought to a halt in order to reduce the chance of data loss.

The end zone is further divided up into additional zones that make up the various memory states, each with an associated threshold as depicted in Figure 4. The calculations are as follows:

- $EmergencyStateThreshold = MaxPhysical - BugCheckZoneSize$.
- $EmergencyZoneSize = [1 - (1/CurrCompRatio)] \times (L3CacheSize + NonPagedPool + LockedPagesCount) + PagerCodeSize$.
- $WarningStateThreshold = EmergencyStateThreshold - EmergencyZoneSize$.
- $WarningZoneSize = EndZoneSize - BugCheckZoneSize - EmergencyZoneSize$.
- $SteadyStateThreshold = WarningStateThreshold - WarningZoneSize$.

Note that the size of the emergency zone is calculated by using the inverse of the compression ratio. Assuming a heterogeneous pattern of data throughout main memory implies that as the compression ratio deteriorates, the data that make up the emergency zone will also share this trait. This means that the opportunity for expansion of the

data will be diminished as well. A smaller emergency zone can be tolerated, with the balance of the space dedicated to the warning zone. An improving compression ratio has more of the end zone given to the emergency zone instead of the warning zone in order to allow for maximum tolerance of data expansion.

CMS passes the thresholds for each of the memory states to the driver, and then registers three event "handles" to be signaled as the system changes its memory state. Once the interrupt thresholds have been calculated, CMS calculates the m_MinConsumptionPhysical value. This variable represents the amount of physical memory that must be in use for CMS to determine whether or not a memory adjustment is necessary. It is set to the size of the emergency zone. The rationale is that as the emergency zone increases in size, CMS has to check earlier for trouble. Note that the zones, thresholds, and m_MinConsumptionPhysical are recalculated periodically as usage of real and physical memory changes. As described in the general introduction to the compression-management software, processes must be prevented from running if the physical-memory compression continues to degrade despite forced paging and zeroing of pages. For this purpose, the CMS spawns and binds one CPU blocker thread per processor in the system. The CPU blockers are utilized when all user applications must be prevented from running. Finally, the CMS spawns a thread in which it executes the compressed-memory management algorithm.

The CMS management thread waits for one of the memory-state notification events from the driver, the terminate event, or a wait timeout value. The wait timeout value varies between the default value (1000 ms) and 0 for any memory state other than steady state. Upon leaving the wait, CMS will gather updated memory usage statistics via a memory-status provider object that is appropriate to the version of Windows on which the code is being executed. One of the more interesting aspects of the memory-status provider is the way it determines the amount of memory that Windows NT/2000 thinks is in use. The operating system provides memory usage statistics through a documented API. However, those statistics are not adequate for controlling compressed memory. In order to make the correct computation, the number of free zeroed pages is needed. While the operating system zeros all free pages in the system, the statistic for available memory includes not only the free-page list (zeroed pages) but also the standby list and the zero-page list (list of pages to be zeroed). There is no documented way to retrieve the number of pages on the free-page list. The solution is to generate a random set of pages to sample across the size of real memory. The corresponding set of CTT entries for each page is examined. The number of zeroed pages in the set is calculated, and the number of zeroed pages in the system is estimated from the results.

The physical-memory usage statistics are modified to include the size of the compression translation table (CTT) and any uncompressed-memory regions that have been defined. Typically the uncompressed-memory region is the lower 1MB DOS area and is set up by the BIOS. This is necessary because the physical-memory usage reported by the memory-compression controller does not account for these values. The m_TZone variable represents the sum of the 32MB L3 cache and the size of the resident portion of the operating system kernel. When the system is in steady state and the physical-memory usage is less than m_MinConsumptionPhysical, any memory held by the memory eaters is released back to the system. The system is considered to be in a safe state. If this is not the case, the TargetedRealMemoryUse is calculated by multiplying the amount of physical memory in use by the boot compression ratio. To determine the memory adjustment needed, the actual amount of real memory in use plus the total amount of memory already held by the memory eaters is subtracted from the TargetedRealMemoryUse. This is called the "adjustment." A negative result means that the compressibility of the system is equal to or greater than the boot compression ratio. This means that the memory eaters should release adjustment units of memory back to the system. If the adjustment is greater than zero, the memory eaters must allocate adjustment units of memory. In doing so, the memory eater calls VirtualAlloc() for the required memory. The memory eater then passes a pointer to the memory and the length to the device driver to perform a zero-page operation on the pages it comprises.

If the CMS returns from its wait state because the warning or emergency event has been signaled, it will switch the timeout for the next iteration to 0. This is done so that the CMS has as much CPU time as possible to analyze the memory conditions and allow the memory eaters to compensate for the deteriorating memory condition as quickly as possible. Recall that there is also one CPU blocker thread per processor waiting for a signal from the driver, indicating that the memory usage has moved into the emergency state. Once the CPU blocker is notified of the emergency condition, it will "busy-spin" on the CPU to which it is bound, which has the effect of blocking all other user applications from running. This is necessary because once the system enters the emergency state, it is very close to running out of physical memory. Allowing user applications to run might further deteriorate memory conditions and lead to system failure. The CPU blocker runs at a real-time-idle priority, which is just below the priority of the CMS and the memory eaters. This permits the compressed-memory management to preempt the blocker threads but also allow the blocker

threads to block other user-mode applications. The CPU blocker threads will stop busy-spinning on the CPUs when they are signaled that the memory system has moved back into the steady or warning state. It is then safe to allow other applications to continue running.

The operating system itself has threads that run at real-time priority, normal or higher. These threads are not a problem for the compression controls because they run for very short durations and cannot change the overall compressibility of the system. However, it is possible for other applications to be run at real-time priority or higher, which in theory can be a problem for the compression controls. However, it should be noted that applications running at real-time priority or higher that do not yield the CPU could interfere with the normal operation of the Windows virtual memory manager itself and cause the operating system to behave erratically. One way to avoid having these applications interfere with the compression-control software is to have the controls dynamically lower the process priority (or suspend the process entirely) while in the emergency state.

Windows 2000 further provides address windowing extensions (AWE) that require special consideration. AWE allows a process to go beyond the 2–3GB user-mode address limitations of the 32-bit Windows/Intel x86 model. Using the AWE API, processes can allocate regions of memory that Windows 2000 will never page to disk. The application creates an "address window" in its process address space into which AWE regions are mapped so that they can be addressed. The compression-management driver provides statistics on AWE usage.

In order to use the AWE API, an application must be running under an account in which the administrator has granted the "pin pages in memory" privilege; and because it is considered a privileged API set, Windows 2000 will allow most of the memory to be pinned down (~2/3 of all memory). Since the pages of memory that are allocated by the virtual memory manager for AWE allocation are not pageable, they cannot be compensated for by using the compression-control method described above. For example, in a 1024MB physical/2048MB real system, an application could allocate ~1400 MB of AWE pages. If the application fills those AWE pages with incompressible data, the system will fail.

Two solutions are currently being examined, both relying on having the device driver trap the AWE system calls and augment their functionality. The most straightforward solution is to treat all AWE regions of memory as if the data that will be written are completely incompressible. For example, half of physical memory can be set aside as the total amount of memory that may be used for AWE in the system. As applications request AWE memory, a check is made to see whether there is enough AWE quota left to grant the request. If so, the allocation request is passed on to the virtual memory manager; otherwise the device driver fails it. CMS has the memory eaters allocate pages to back AWE memory that has been allocated. For example, in the system mentioned above, 512 MB would be allowed to go to AWE pages. This leaves a total of 1024 MB of real pages left to other applications in the system. CMS would have the memory eaters hold 512 MB of real space to back the AWE allocation. A second possibility is to back only the mapped AWE pages. The only way that the compressibility of an AWE region can change is for it to be mapped into the virtual address space of a process. This implementation allows for more AWE memory use, because it does not assume that each region will have the absolute worst compressibility of 1:1.

Reference [6] discusses the need to monitor the I/O request in the MXT architecture. The need arises from the fact that the I/O is streamed through the L3 cache. If too many I/O requests are outstanding, they can force L3 cache writebacks. If these are not properly accounted for, they can result in physical-memory exhaustion when the L3 content is not sufficiently compressible. In the Windows system, a set of filter drivers may be inserted to monitor and throttle the I/O requests.

4. Performance evaluation

Performance impact of compression

The MXT memory system uses a relatively long 1KB compression block size to be able to compress efficiently, since shorter blocks may not compress well. Because of the compression and decompression operations performed on these blocks in the memory controller, memory access times are longer than usual. The 32MB L3 cache contains uncompressed (1KB) lines to reduce the effective access times by locally serving most of the main-memory requests. Since this type of memory organization is new, we used a database benchmark to measure its performance impact. In these experiments, we used an MXT system with dual 733-MHz Pentium III processors with Windows 2000 and a single disk drive. The MXT hardware was an early prototype, which, because of hardware bugs, had some of its performance-enhancing features, such as bus-defer response and processor IOQ depth limited to 1, disabled. We compared the MXT hardware to a standard system with similar hardware characteristics except with no compression or L3 support. We present the results of this database study in the next section. Performance results using CPU benchmarks may be found in [14].

Database benchmark results

The MXT system has been measured running an insurance company database schema. This configuration is used

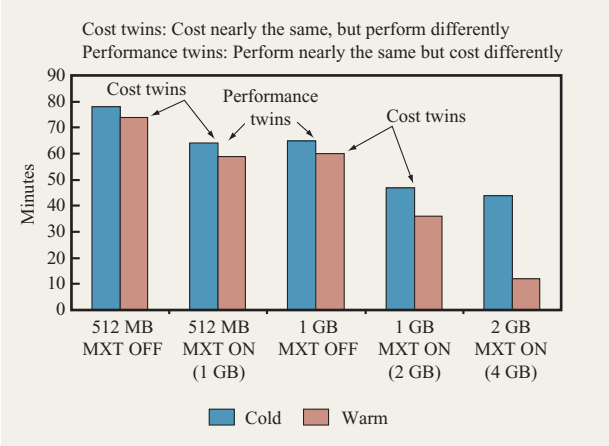

Figure 7

Database benchmark results for five different configurations.

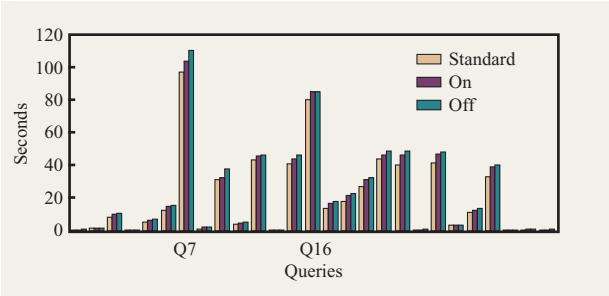

Figure 8

DB2 query run times.

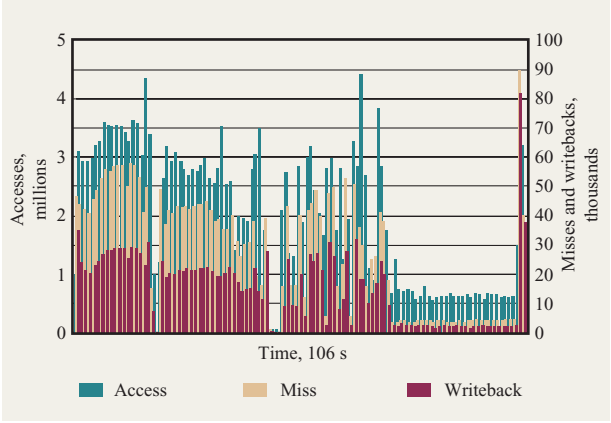

Figure 9

Query 7 L3 accesses vs. misses.

within IBM primarily 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 only 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 (1GB expanded) with MXT turned on, 1 GB MXT off, 1 GB MXT on (2 GB expanded), and 2 GB MXT on (4 GB expanded) configurations. Benchmarks ran on the Windows 2000 operating system. Two runs were made for each configuration: a cold run in which the file cache is initially empty, and immediately following that a second, warm run, in which the buffers have been "warmed" by the preceding cold run.

Figure 7 shows the performance benefits of MXT. For the 512MB 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 1GB 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, which is indicative that both the smaller 512MB memory and 1GB memory configurations are 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 7 also shows "performance twins" and "cost twins" to emphasize the benefits of MXT. Performance twins perform nearly identically; however, the MXT-on twin costs less, since its memory requirements are reduced by one half. Cost twins have the same amount of physical memory, but the MXT-on twin performs better because of the doubling of memory.

Figure 8 shows run times of the individual DB2* queries in a 4GB system after warmup. 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 9** and **10**, 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 with no compression or L3 support. On this early MXT prototype, performance-enhancing features

of the processor bus, such as bus defer response and IOQ depth limited to 1, were disabled because of hardware bugs, which is one possible explanation. Another possibility is that higher L3 miss rates degrade overall performance compared to a standard system without an L3 cache.

Compressibility of applications

Now that the performance of the MXT system has been established, we turn our attention to the compressibility of main-memory content for various applications. We measured the compression ratios on the actual MXT hardware whenever possible, and we used an estimation tool when MXT hardware was not available. The estimation tool samples the live memory content while the application is running on a standard computer and predicts the compression ratio. Results show that for most of these applications, main-memory content 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. Every two seconds, a sampler program reads 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 memory utilization of the benchmark application, 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 postprocessing step, we took the average of the samples to produce the average compression ratio of a given benchmark.

Figure 11 shows the compression ratios for a few applications. Synopsis, Photoshop**, MSDN Install, and DB2 compression ratios were measured on the MXT hardware with the Windows NT 4.0 or Windows 2000 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

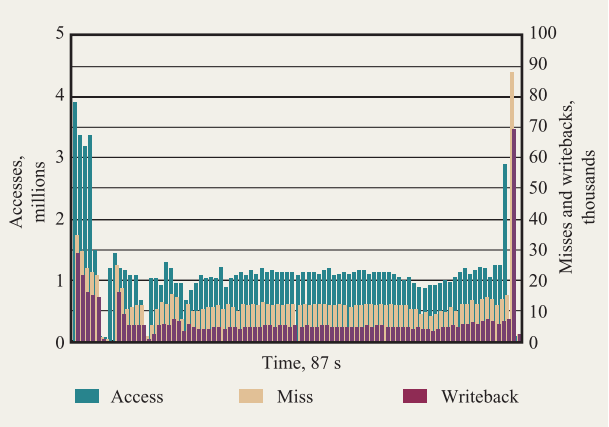

Figure 10

Query 16 L3 accesses vs. misses.

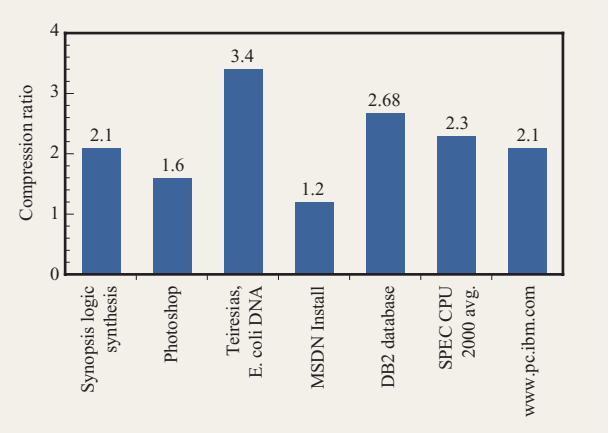

Figure 11

Compressibility of various applications.

most software installations compress poorly, since the CD-ROM files are already compressed. The install program itself consumes only 4 MB. However, the associated file cache or Windows 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 twelve integer benchmarks in the SPEC suite. The web site *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 12 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 and was a cold run, reading the database from disk. The second and

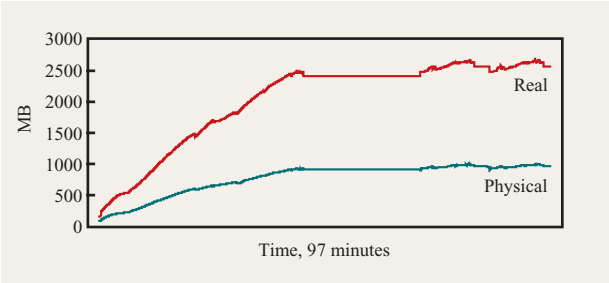

Figure 12

DB2 database compressibility: real vs. physical memory utilization.

third runs took 12 minutes each, following 20 minutes of idle time. The average compressibility was 2.68:1.

5. Conclusions

In this paper we have described and evaluated a computer system with hardware main-memory compression that effectively doubles the size of the main memory. We presented an overview of the MXT hardware technology and then described the software support for main-memory compression under Linux and Windows NT/2000. We measured the impact of compression on application performance and determined that hardware compression has a negligible penalty over an uncompressed hardware. We measured real and physical memory utilization for a few applications and determined that main-memory content can usually be compressed by a factor of 2 or more.

Future work might include a more detailed study of the L3 cache shared by CPUs and I/O devices. L3 performance with several applications running at the same time or with I/O streaming through the L3 cache will require further study.

Acknowledgments

The authors express thanks to Brett Tremaine and Michael Wazlowski, who described the operation of the L3 cache/compressed-memory controller chip to us, and to Chuck Bauman, Peter Franaszek, Michel Hack, and Philip Heidelberger for many valuable discussions and suggestions during this project. Parts of this paper were presented at the Memory Wall Workshop [4]; we are grateful to the referees of this paper as well as the workshop organizers, referees, and attendees for their valuable comments and criticism.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Microsoft Corporation, Intel Corporation, The Open Group, Transaction Processing Performance Council, or Adobe Systems, Inc.

References

1. D. A. Luick, J. D. Brown, K. H. Haselhorst, S. W. Kerchberger, and W. P. Hovis, "Compression Architecture for System Memory Applications," U.S. Patent 5,812,817, 1998.
2. P. Franaszek, J. Robinson, and J. Thomas, "Parallel Compression with Cooperative Dictionary Construction," *Proceedings of the Data Compression Conference, DCC'96*, IEEE, 1996, pp. 200–209.
3. S. Arramreddy, D. Har, K. Mak, T. B. Smith, B. Tremaine, and M. Wazlowski, "IBM X-Press Memory Compression Technology Debuts in a ServerWorks NorthBridge," presented at the HOT Chips 12 Symposium, August 13–15, 2000.
4. B. Abali and H. Franke, "Operating System Support for Fast Hardware Compression of Main Memory," presented at the Memory Wall Workshop, International Symposium on Computer Architecture (ISCA2000), Vancouver, B.C., July 2000.
5. P. Franaszek and J. Robinson, "Design and Analysis of Internal Organizations for Compressed Random Access Memory," *Research Report RC-21146*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, April 1998.
6. P. Franaszek, P. Heidelberger, and M. Wazlowski, "On Management of Free Space in Compressed Memory Systems," *Proceedings of the ACM Sigmetrics Conference*, ACM, Atlanta, GA, June 1999, pp. 113–121.
7. P. Wilson, S. Kaplan, and Y. Smaragdakis, "The Case for Compressed Caching in Virtual Memory Systems," *Proceedings of the USENIX Annual Technical Conference*, USENIX Association, Monterey, CA, June 1999, pp. 6–11.
8. M. Kjelso, M. Gooch, and S. Jones, "Empirical Study of Memory Data: Characteristics and Compressibility," *IEE Proceedings on Computers and Digital Techniques*, Vol. 45, No. 1, pp. 63–67, IEE, 1998.
9. Sergei Y. Larin and Thomas M. Conte, "Compiler-Driven Cached Code Compression Schemes for Embedded ILP Processors," *Proceedings of the Annual International Symposium on Microarchitecture*, 1999, pp. 82–92.
10. I.-C. K. Chen, J. T. Coffey, and T. N. Mudge, "Analysis of Branch Prediction via Data Compression," *Computer Architecture News* **24,** 128–137 (October 1996).
11. John Kalamatianos and David R. Kaeli, "Predicting Indirect Branches via Data Compression," *Proceedings of the Annual International Symposium on Microarchitecture*, 1998, pp. 272–281.
12. P. A. Franaszek, P. Heidelberger, D. E. Poff, and J. T. Robinson, "Algorithms and Data Structures for Compressed-Memory Machines," *IBM J. Res. & Dev.* **45,** No. 2, 245–258 (2001, this issue).
13. U. Vahalia, *UNIX Internals, The New Frontiers*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1996, ISBN 0-13-101908-2.
14. B. Abali, H. Franke, D. E. Poff, X. Shen, and T. B. Smith, "Performance of Hardware Compressed Main Memory," *Proceedings of the Seventh International Symposium on High Performance Computer Architecture (HPCA-7)*, Monterrey, Mexico, January 20–24, 2001, pp. 73–81.

Received August 29, 2000; accepted for publication March 5, 2001

Bulent Abali *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (abali@us.ibm.com).* Dr. Abali has been a Research Staff Member at the IBM Thomas J. Watson Research Center since 1989; he is currently a manager responsible for system software and performance evaluation of advanced memory systems. He has contributed to numerous projects on parallel processing, high-speed interconnects, and memory systems, including RS/6000 SP and MXT. Dr. Abali received his Ph.D. degree in electrical engineering from Ohio State University.

Hubertus Franke *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (frankeh@us.ibm.com).* Since 1993 Dr. Franke has been a Research Staff Member at the IBM Thomas J. Watson Research Center, where he currently manages the Enterprise Linux group. He contributed to the IBM SP2 system software and architecture, the K42 operating system, the development of Linux for highly scalable architectures, and the MXT Linux support. Dr. Franke has received multiple IBM Outstanding Innovation Awards and Outstanding Technical Achievement Awards, and he has published more than 50 papers and 13 patents. He received a first-in-class Diplom. in Informatik from the Technical University of Karlsruhe, Germany, in 1987 and a Ph.D. degree in electrical engineering from Vanderbilt University in 1992. His research interests include architectures and operating systems for highly scalable systems, distributed systems, and system security. He is a member of the IEEE.

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

Robert A. Saccone, Jr. *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (rsaccone@us.ibm.com).* Mr. Saccone is a Senior Software Engineer at the IBM Thomas J. Watson Research Center, where he designs and develops software to enable operating systems to take advantage of MXT technology. Before joining the Watson Research Center in 1999, he was a lead engineer and manager at Symantec Corporation, Framingham/Bedford, Massachusetts, where he worked on the rewrite of pcANYWHERE for the Windows NT/Windows 9X platforms. He has also held positions at Cheyenne Software, Rolling Meadows, Illinois, and Computer Associates, Lisle, Illinois, where he focused on network management software and rapid application development tools. His areas of interest are large-scale software design, distributed systems, and operating systems implementations. Mr. Saccone received his B.S. degree in computer science from the State University of New York at Stony Brook in 1989. He has three patents pending.

Charles O. Schulz *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (cschulz@watson.ibm.com).* Mr. Schulz is a manager at the IBM Thomas J. Watson Research Center, where he is responsible for advanced memory and systems architectures supporting the IBM xSeries products. He received his B.S. and M.S. degrees in electrical and electronic engineering from North Dakota State University in 1971 and 1972, respectively. Prior to joining IBM in 1990, he held engineering and management positions at various aerospace and computer companies. Mr. Schulz has extensive experience in high-reliability and fault-tolerant computer design as well as computer design for real-time control of aircraft and critical aircraft systems. His current research interests include computer architecture for high-performance scalable and partitioned servers. He has one issued patent and fifteen pending, as well as various technical publications on computer architecture and design.

Lorraine M. Herger *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (herger@us.ibm.com).* Ms. Herger joined the IBM Research Division in 1990. She is currently a Senior Technical Staff Member and Senior Manager of the Emerging Systems Software group, which explores new directions in operating environments. Before joining the Watson Research Center, she was a design and development engineer at the IBM Kingston, New York, laboratory and IBM Instruments, contributing to several IBM products. Ms. Herger received an M.B.A. degree from the Stern School, New York University, in 2000, and her B.S.E.E. degree from the University of Maryland before joining IBM in 1982.

T. Basil Smith *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (tbsmith@us.ibm.com).* Dr. Smith has been a Research Staff Member at the IBM Thomas J. Watson Research Center since 1986. He is currently a Senior Manager responsible for research into exploitation of high-leverage server innovations and manages the Open Server Technology Department. His work has been on memory hierarchy architecture, reliability, durability, and storage efficiency enhancements in advanced servers. Dr. Smith has received IBM Outstanding Innovation Awards and Outstanding Technical Achievement Awards for his contributions in these fields at IBM. Before joining IBM in 1986, he worked at United Technologies Mostek Corporation in Dallas and at the Charles Stark Draper Laboratory in Cambridge, Massachusetts. He holds more than 20 patents in computer architecture and reliable machine design. Dr. Smith received his Ph.D. degree in computer systems, and his S.M. and S.B. degrees from MIT. He is an IEEE Fellow and a member of the IEEE Computer Society Technical Committee on Fault-Tolerant Computing, and is active in that community. Most recently he was General Chair of the Dependable Systems and Networks Conference (DSN-2000) held in New York in June 2000.