IBM Memory Expansion Technology (MXT)

by R. B. Tremaine
P. A. Franaszek
J. T. Robinson
C. O. Schulz
T. B. Smith
M. E. Wazlowski
P. M. Bland

Several technologies are leveraged to establish an architecture for a low-cost, high-performance memory controller and memory system that more than double the effective size of the installed main memory without significant added cost. This architecture is the first of its kind to employ real-time main-memory content compression at a performance competitive with the best the market has to offer. A large low-latency shared cache exists between the processor bus and a content-compressed main memory. High-speed, low-latency hardware performs real-time compression and decompression of data traffic between the shared cache and the main memory. Sophisticated memory management hardware dynamically allocates main-memory storage in small sectors to accommodate storing the variable-sized compressed data without the need for "garbage" collection or significant wasted space due to fragmentation. Though the main-memory compression ratio is limited to the range 1:1–64:1, typical ratios range between 2:1 and 6:1, as measured in "real-world" system applications.

Introduction

Memory costs dominate both large memory servers and expansive computation server environments such as those employed in today's "data centers" and "computer farms." These costs are both fiscal and physical (e.g., volume, power, and performance associated with the memory system implementation), and often aggregate to a significant cost constraint that the information technology (IT) professional must trade off against computing goals.

Data compression techniques are employed pervasively throughout the computer industry to increase the overall cost efficiency of storage and communication media. However, despite some experimental work [1, 2], system main-memory compression has not been exploited to its potential. IBM Memory Expansion Technology (MXT*) addresses the system memory cost issue with a new memory system architecture that more than doubles the effective capacity of the installed main memory without significant added cost.

MXT is directly applicable to any computer system, independent of processor architecture or memory device technology. MXT first debuted in the Serverworks "Pinnacle" chip, an Intel Pentium** III/Xeon** bus-compatible, low-cost single-chip memory controller (Northbridge) [3]. This unique chip is the first commercially available memory controller to employ real-time main-memory content compression at a performance level competitive with those of the market's best products.

Architecture

Conventional "commodity" computer systems typically share a common architecture, in which a collection of processors are connected to a common SDRAM-based

0018-8646/01/$5.00 © 2001 IBM

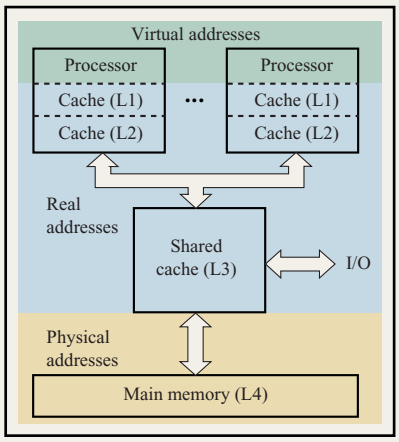

Figure 1

System memory hierarchy.

main memory through a memory controller chip set. MXT incorporates the two-level main-memory architecture shown in **Figure 1**, consisting of a large shared cache coupled with a typical main-memory array. The high-speed cache, containing the frequently referenced processor data, architecturally insulates the overall system performance from access latency to the main memory, thereby opening opportunities for trading off increased memory-access latency for greater function. For example, remote/distributed, very large and/or highly reliable features may be incorporated without adverse effects on overall system performance.

The shared cache, coupled with the recent advent of high-density 0.25-μm and smaller-geometry ASIC technology, is leveraged to incorporate a new "compressed" main-memory architecture. Special logic-intensive compressor and decompressor hardware engines provide the means to simultaneously compress and decompress data as it is moved between the shared cache and the main memory. The compressor encodes data blocks into as compact a result as the algorithm permits. A sophisticated memory management architecture is employed to permit storing the variable-sized compressed data units in main memory, while mitigating fragmentation effects and avoiding "garbage collection" schemes. This new architecture serves to halve the cost of main memory without any significant degradation in overall system performance.

The internal structure for a typical single-chip memory controller with an external cache memory and on-chip cache directory is shown in **Figure 2**. Any processor or I/O memory references are directed to the cache controller, resulting in cache directory (dir) lookup to determine whether the address is contained within the cache. Cached references are serviced directly from the cache, while cache read misses are "deferred," and the least-recently-used cache line is selected for replacement with the new cache line that contains the requested address. The cache controller issues a request for the new cache line from the main-memory controller, while at the same time writing the old cache line back to the writeback buffer (wtq) in cases in which the old cache line contains modified data.

To service the new cache-line fetch, the memory controller first reads a small address-translation table entry from memory to determine the location of the requested data. Then the memory controller reads the requested data. Data is either streamed around the decompressor (decomp) when uncompressed, or through the decompressor when compressed. In either case, the data is then streamed through the elastic buffer (rdq) to the cache. The memory controller provides seven-cycle advance notification of when the requested 32-byte (32B) data will be in the critical word (cw) buffer. This enables the processor bus controller to arbitrate for a deferred read reply to the processor data bus, and deliver data without delay.

Cache writeback activity is processed in parallel with read activity. Once an entire cache line is queued in the writeback buffer, the compression commences and runs uninterrupted until it is complete, 256 cycles later. Then, when a spatial advantage exists, the memory controller stores the compressed data; otherwise, the memory controller stores the uncompressed writeback data directly from the writeback buffer. In either case, the memory controller must first read the translation table entry for the writeback address in order to allocate the appropriate storage and update the entry accordingly before writing it back to memory. The data itself is then written to memory within the allocated sectors.

Shared-cache subsystem

The shared cache provides low-latency processor and I/O subsystem access to frequently accessed uncompressed data. The data-code–I/O unified cache content is always uncompressed and is typically accessed at 32B granularity. Write accesses smaller than 32B require the cache controller to perform a read–modify–write operation for the requesting agent. The cache is partitioned into a quantity of *lines*, with each line an associative storage unit equivalent in size to the 1KB uncompressed data block size. A cache directory is used to keep track of real-memory *tag* addresses which correspond to the cached addresses that can be stored within the line, as well as any relevant coherency management state associated with the cache line.

The shared cache can be implemented in any of three primary architectures, where performance can be traded off with the cost of implementation:

1. The independent cache array scheme [4] provides the greatest performance, but at the highest cost. The large independent data-cache memory is implemented using low-cost double-data-rate (DDR) SDRAM technology, outside the memory controller chip, while the associated cache directory is implemented on the chip. The cache size is limited primarily by the size of the cache directory that can fit on the die. For example, a single-chip memory controller implemented in 0.25-μm CMOS can support a 32MB cache, whereas a 64MB cache can be supported in 0.18-μm CMOS, assuming a 12-mm die. The additional hardware cost for this type of implementation ranges between $50 and $60. However, the performance is maximized because the cache interface can be optimized for lowest-latency access by the processor, and the main-memory interface traffic is segregated from the cache interface.
2. The compressed main-memory cache partition scheme [1] involves logically apportioning an uncompressed cache memory region from the main memory. The cache controller and the memory controller share the same storage array via the same physical interface. Data is shuttled back and forth between the compressed main-memory region and the uncompressed cache through the compression hardware during cache-line replacement. An advantage to this scheme is that the compressed cache size can be readily optimized to specific system applications. The additional hardware cost can range from $0 to $30, and is most advantageous when the cache directory is stored in a main memory partition as well. Performance is particularly disadvantaged by contention for the main-memory physical interface by the latency-sensitive cache controller.
3. The managed number of uncompressed cache lines within the compressed-memory scheme caches uncompressed data implicitly. Rather than apportion a specific uncompressed cache region from the main-memory storage, the cache is distributed throughout the compressed memory as a number of uncompressed lines. Only the most-recently-used n lines are selected to make up the cache. Data is shuttled in and out of the compressed memory, changing the compressed state as it is passed through the compression logic during cache-line replacement. An advantage to this scheme is that no separate cache directory is required, since the compressed-memory sector translation table (STT) serves in this capacity, thus permitting a much larger cache. However, performance benefits can be obtained from a small hardware STT cache to maintain rapid access to the recently used STT entries pertaining to cached blocks. Another advantage is that the effective cache size may be dynamically optimized during system operation by simply changing the maximum number (n) of uncompressed lines. Performance is disadvantaged by contention for the main-memory physical interface, as well as a greater average access latency associated with the cache directory references. Further, since the cache lines are not directly mapped, as is the case in a conventional cache structure, any on-chip directory must include enough information to locate the cache lines, rendering the directory less spatially efficient.

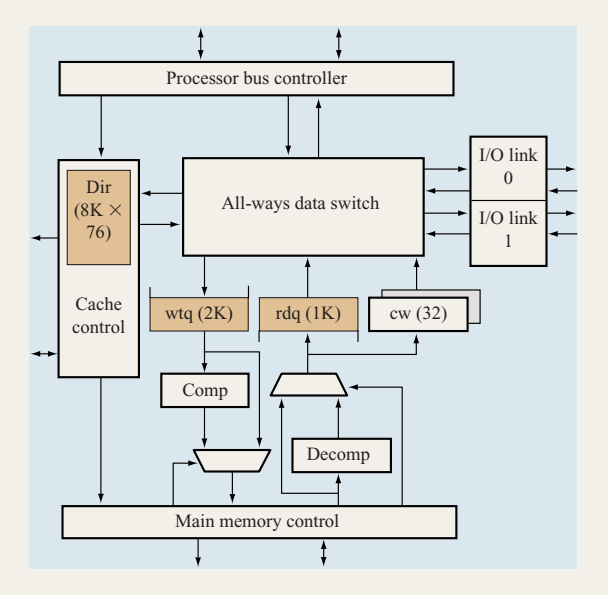

Figure 2

Typical control chip block diagram.

For any case, the relatively long cache line merits special design consideration. For example, processor reference bits are used to mitigate extraneous cache-coherency snoop traffic on the processor bus. These bits are used to indicate when any processor has referenced one or more cache-line segments. When a cache line is evicted, only "referenced" cache-line segments, instead of the entire cache line, need be invalidated on the processor bus.

Shuttling the wide lines in and out of the cache during cache-line replacement requires many system clock cycles, typically at least 64 cycles for each writeback and line-fill operation. To alleviate processor access stalls during the lengthy cache-line replacement, the cache controller permits two logical cache lines to coexist within one physical cache line. This mechanism permits the cache

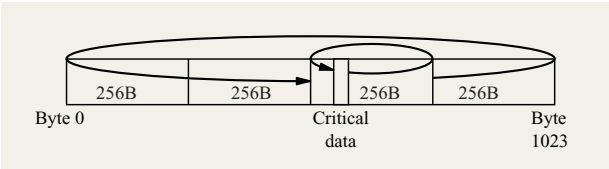

Figure 3

Critical read request data fetch order.

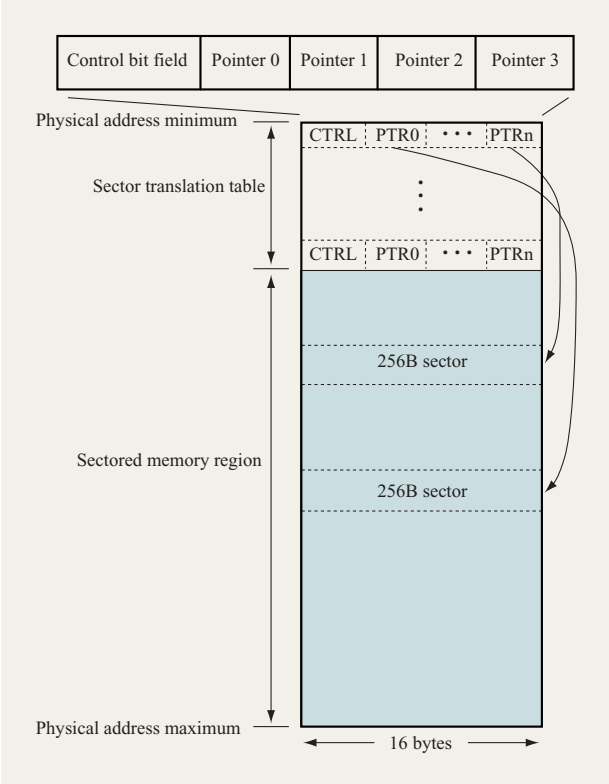

Figure 4

Memory organization.

line to be written back, reloaded, and referenced simultaneously during cache-line replacement.

During replacement, a state vector is maintained to indicate an *old*, *new*, or *invalid* state for each of *n* equal subcache-line units within the physical line. As subcache lines are invalidated or moved from the cache to the writeback buffer, they are marked *invalid*, indicating that the associated new subcache-line data may be written into the cache. Each time a new subcache line is loaded, the associated state is marked *new*, indicating that processor or I/O access is permitted to the new subcache-line address. Further, processor and I/O accesses to the old cache address are also permitted when the associated subcache lines are marked *old*. Cache lines are always optimally fetched and filled, so that the subcache-line writeback follows the same subcache-line order to maximize the number of valid cache lines at all times.

The cache supports at least two concurrent cache-line replacements. Two independent 1KB writeback buffers (wtq) facilitate a store-and-forward pipeline to the main memory, and one 1KB elastic buffer (rdq) queues line-fill data when the cache is unavailable for access. A writeback buffer must contain the entire cache line before the main-memory compressor may commence the compression operation. Conversely, the line-fill data stream is delivered directly to the cache as soon as a minimum subcache-line unit of data is contained within the buffer. Two independent 32B *critical word* (CW) buffers are used to capture the data associated with cache misses for direct processor bus access.

Main-memory subsystem

The main-memory subsystem stores and retrieves cache lines in response to shared-cache writeback (write) and line-fill (read) requests. Data is stored within the main-memory array, which comprises industry-standard SDRAM dual in-line memory modules (DIMMs). The memory controller typically supports two separate DIMM configurations for optimal application in both large and small server applications. The *direct-attach* configuration supports a few single/double-density DIMMs directly connected to the memory controller chip(s) without any "glue" logic. The *large-memory* configuration supports one or more cards with some synchronous memory rebuffering chips connected between the controller and the memory array. In either configuration, the array is accessed via a high-bandwidth interface with 32B–256B access granularity. For minimal latency, uncompressed data references are always retrieved with the critical granule first, and 256B addresses wrapped as shown in **Figure 3**.

The main-memory subsystem may be configured to operate with compression disabled, enabled for specific address ranges, or completely enabled. When compression is disabled, the physical memory address space is directly mapped to the real address space in a manner equivalent to conventional memory systems. Otherwise, the memory controller provides real-to-physical address translation to accommodate dynamic allocation of storage for the variable-sized data associated with compressed 1KB lines. The additional level of address translation is carried out completely in hardware, using a translation table apportioned from the main memory.

The physical memory is partitioned into two regions, or optionally three when uncompressed memory is configured. The memory comprises two primary data structures: the *sector translation table* (STT) and the *sectored memory* (**Figure 4**). The STT consists of an array

of 16B entries in which each entry is directly mapped to a corresponding 1KB real address. Therefore, the number of STT entries is directly proportional (1/64) to the size of the real address space[1] declared for a given system. Each entry describes the attributes for the data stored in the physical memory and associated with the corresponding 1KB address. Data may occur in one of three conditions:

- Compressed to ≦120 bits.
- Compressed to >120 bits.
- Uncompressed.

When a 1KB data block is compressible to less than 121 bits, the data is stored directly into the STT entry with appropriate flags, yielding a maximum compressibility of 64:1. Otherwise, the data is stored outside the entry in one to four 256B sectors, with the sector pointers (PTR) contained within the STT entry as shown in **Table 1**. For the case in which the data block is uncompressed, four sectors are used, and the STT entry control field indicates the "uncompressed" attribute. In cases in which unused "fragments" of sector memory exist within a 4KB real page, any new storage activity within the same page can share a partially used sector in increments of 32B. A maximum of two 1KB blocks within a page may share a sector. This simple two-way sharing scheme typically improves the overall compression efficiency by 15%, with nearly all of the potential gain attainable from combining fragments with any degree of sharing [5].

The sectored memory consists of a "sea" of 256B chunks of storage, or sectors, that are allocated from a "heap" of free sectors available within the sectored memory region. The heap is organized as a linked list of unused sector addresses, with the list head maintained within a hardware register. The list itself is stored within the free sectors, so the utilization of sectors oscillates between holding the free list and data. As shown in **Figure 5**, each node of the free list contains pointers to 63 free 256B sectors and one pointer to the next 256B node in the free list. Since the node is itself a free or unused 256B sector, the free list effectively requires no additional storage.

[1] The real address space is defined to the operating environment through a hardware register. The BIOS firmware initializes the register with a value based on the number and type of DIMMs installed in a system. When compression is enabled, the BIOS doubles this value to indicate a real address space twice as large as that populated with DIMMs.

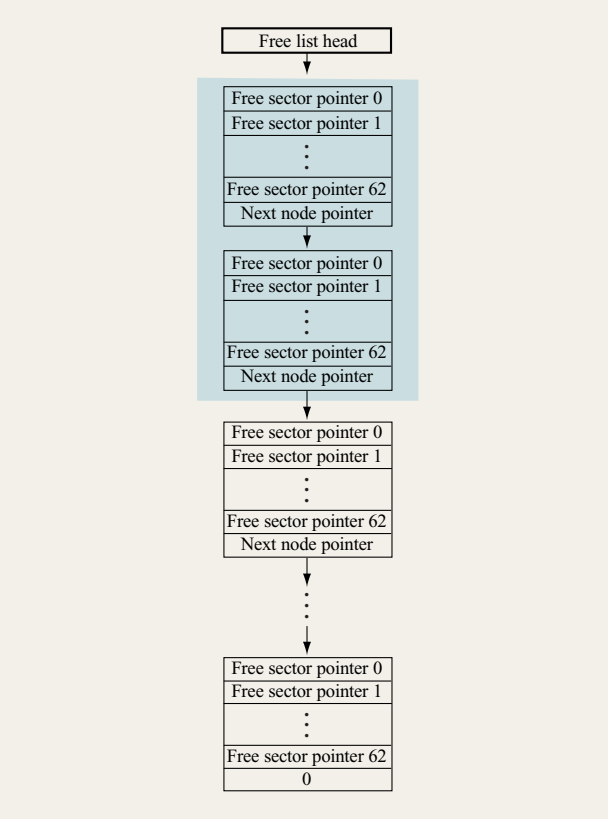

Figure 5

Free list.

A small hardware cache contains the leading two nodes (shaded in Figure 5) of the free list, for rapid access during allocation and deallocation of sectors associated with data storage requests.

Uncompressed memory regions are areas of the real address space in a compressed memory in which the data is never compressed. These regions are configurable as a 32KB-aligned 32KB–256MB range. They are apportioned from the top of the sectored memory and are direct-mapped as shown in **Figure 6**. The access latency to these regions is minimized because data is directly addressable without the intermediate step of referencing an STT entry. Of course, the data is fetched with the requested 32B first, as is always the case for uncompressed data.

Table 1 Sector translation table (STT) entry encoding.

127–125	*124*	*123–122*	*121–120*	*119–0*			
7	0	<0, P>	class<1:0>	Compressed line code			
size <2:0>	E	reserved	class<1:0>	PTR4<37:8>	PTR3<37:8>	PTR2<37:8>	PTR1<37:8>

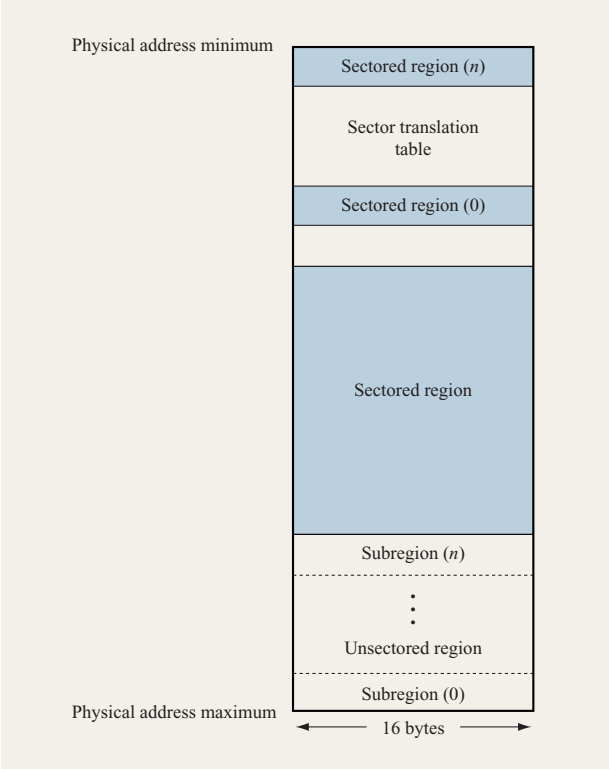

Figure 6

Unsectored memory organization.

The regions within the STT that contain entries for addresses within unsectored regions are never referenced. Not to be wasted, these "holes" within the STT are made available as additional sectored storage through incorporation into the free list.

Additional storage efficiency can be obtained when the shared-cache implementation employs an *inclusive* cache policy whereby the cache levels closer to the CPU always contain a subset of data contained in those further away. This implies that any modified line within the shared cache is more recent than the original copy located within the compressed main memory. The memory sectors used to store this "stale" data can be recovered and returned to the free list for use in storing other information. This recovery process requires the hardware to reference the STT entry associated with the cache line, deallocate any associated sectors, and then copy the updated STT entry back to the memory. The process occurs when the shared-cache controller notifies the memory controller that a cache line has been placed in the *modified* state—for example, when the cache is fetching a line from the main memory to satisfy a *write* or *read with intent to modify* (RWITM) cache access.

Page operations

A beneficial side effect of "virtualizing" the main memory through a translation table is that simple alteration of a table entry can be used to relocate and/or clear data associated with the entry. We capitalized on this notion by implementing a programmed control mechanism to enable real memory page (4KB) manipulation, at speeds ranging between 0.1 and 3.0 μs, depending on the amount of processor bus coherency traffic required. We also provide a means for programmed "tagging" of pages, using a *class* field within the sector translation table entry reserved for this purpose. Special hardware counters are employed to count the allocated sectors by class. This scheme provides a means for software to establish metrics for types or classes to support algorithms for memory usage optimization.

Page operations are initiated atomically to the memory controller; i.e., the controller will not accept a new page request until all pending requests are completed. This ensures coherency when more than one processor may be attempting to request page operations. The page-request mechanism requires a programmable *page* register with the page address and command bits, and a *page complement* register for specifying the second page address for commands involving two separate pages. The following page commands are supported by the architecture:

0000 No operation.

0001 Clear (invalidate) 4KB page from the system memory hierarchy.

0010 Flush–invalidate 4KB page from memory hierarchy above system memory.

0011 Transfer *Class* field to the address STT entry.

0100 Move to complement page and zero source. Sequence: 1) Flush–invalidate the 4KB page from the memory hierarchy above the physical memory. 2) Invalidate the 4KB complement page from the memory hierarchy above the physical memory. 3) Swap the STT entry with the complement page STT entry.

0101 Move to complement page and zero source without coherency. Sequence: 1) Flush–invalidate the 4KB page from the memory hierarchy above the physical memory. 2) Invalidate the 4KB complement page in physical memory without regard to the memory hierarchy state. 3) Swap the STT entry with the complement page STT entry.

0110 Swap with complement page. Sequence: 1) Flush–invalidate the 4KB page from the memory hierarchy above the physical memory. 2) Flush–invalidate the 4KB complement page from the memory hierarchy above the physical memory. 3) Swap the STT entry with the complement page STT entry.

0111 Swap with complement page without coherency. Sequence: 1) Flush–invalidate the 4KB page from the memory hierarchy above the physical memory. 2) Swap the STT entry with the complement page STT entry.

1000 Move page STT to a software-accessible STT-entry hardware register.

1001 Invalidate 4KB page from the system memory hierarchy and transfer *Class* field to the STT entry.

other Reserved.

Compression/decompression

The compression/decompression mechanism is the cornerstone of MXT. Compression, as applied in the main-memory data flow application, requires low latency and high bandwidth in the read path, and of course it must be lossless. Although a plethora of compression algorithms exist, none met our architectural criteria. We chose to leverage the recently available high-density (0.25-μm) CMOS ASIC technology by implementing a gate-intensive, parallelized derivative [6] of the popular Ziv–Lempel (LZ77) "adaptive dictionary" approach. With this new scheme, the unencoded data block is partitioned into n equal parts, each operated on by an independent compression engine, but with shared dictionaries. It has been shown experimentally that parallel compressors with cooperatively constructed dictionaries have compression efficiency essentially equivalent to that of the sequential LZ77 method [6].

Typically, four compression engines are employed, each operating on 256 B (one quarter of the 1KB uncompressed data block), at the rate of 1B/cycle, yielding a 4B/cycle (or 8B/cycle when double-clocked) aggregate compression rate. **Figure 7** shows the four compression engines, each containing a history buffer or *dictionary* comprising a 255B content-addressable memory (CAM) that functions as a shift register. Attached to each dictionary are four 255B comparators for locating the incoming reference byte within the entire dictionary structure. During each clock cycle, one byte from each 256B source data block (read from the shared-cache writeback buffer) is simultaneously shifted into a respective dictionary and compared to the accumulated (valid) dictionary bytes. The longest match of two or more bytes constitutes a working string, while the copy in the dictionary is the reference string. Should a single byte match or no match be found, as may be the case for random data, the reference byte is a *raw character*.

Compression occurs when working strings within the compare data stream are replaced with location and length encoding to the reference strings within the dictionary. However, it can be seen in **Table 2** that the *raw-character* encoding scheme may result in a 256B uncompressed data stream actually expanding to 288 bytes for a given engine. Therefore, special detection logic is employed to detect the point at which the accumulated aggregate compressed output exceeds 1 KB (or a programmed threshold), causing compression to be aborted and the uncompressed data block to be stored in memory.

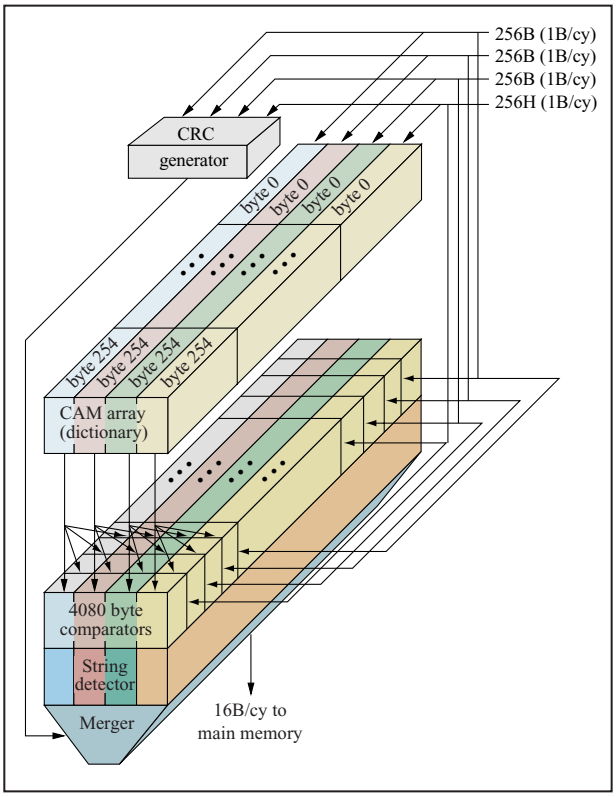

Figure 7

Compressor block diagram.

Table 2 Compression encoding.

Compressed data type	*Encoding*
Raw character	{0, data byte}
String	{1, primary length, position, secondary length}

Strings are detected by one of 255 detectors from any one of the four dictionaries. Once an emerging string is detected, future potential strings are ignored until the end of the current string is detected. The string detector calculates the length and position of a working string. At the end, only the longest string, or that string starting nearest to the beginning of the dictionary for multiple strings with the same length, is selected. The length field ranges from two to twelve bits to encode the number of

bytes in the working string, using a Huffman coding scheme. The position field ranges from two to ten bits to encode the starting address of the reference string. Merge logic is responsible for packing the variable-length bit stream into a word-addressable buffer.

Computer systems that employ hardware memory compression may at times encounter significant processor stall conditions due to compressor write-queue "full" conditions. The MXT architecture provides a means to abort a pending compression operation for the purpose of writing the data directly to the main memory, bypassing the compressor hardware during stall conditions. Memory space (compressibility) is sacrificed for higher system performance during these temporary writeback stall events. A background memory scrub later detects and recovers the "lost" compressibility by recycling the uncompressed data back through the compressor during idle periods.

The much simpler decompressor comprises four engines, each decoding the encoded compressed data block. Each engine can produce 2B/cycle, yielding an aggregate 8B/cycle when 1X-clocked or 16B/cycle when 2X-clocked, as occurs in Pinnacle.

Power-down state

Computer system main-memory capacity is growing significantly faster than the means to transfer the content to nonvolatile (NV) storage (magnetic disk). This relationship is particularly relevant when a system encounters a power outage and there is insufficient reserve power to permit the system to copy the modified memory content to NV storage. An MXT architectural extension provides a means to rely on an NV main memory implementation in lieu of copying information back to magnetic disk during power outages. A low-power "sleep" state, in which only the SDRAM memory remains powered by a backup power source, is adequate for nearly all power outages. Even a limited NV memory system can support extended outages (greater than 24 hours) by providing time for remedial action to restore or provide additional backup power.

A 2KB *system state* region is reserved between the bottom (lowest-order address) of physical memory and the STT region. This region is "shadowed" behind the real address space, such that it is not "seen" by application or operating system programs when not enabled, since it is reserved for exclusive use by the machine's built-in operating system (BIOS) or system management program software. A special programmable hardware control bit is used to toggle the physical main-memory address space between the shadowed system state memory and the normal physical address space. When the system state memory is enabled, the memory controller aliases all memory references to the region (i.e., the system appears to have a 2KB memory with compression disabled).

Upon detection of an impending power loss, software can store the system hardware and software state in this region after quieting the system and flushing the cache hierarchy to the main memory. The system state information will be retained through a power outage that affects all hardware except the NV main memory. After power is restored, the BIOS software reestablishes the memory configuration for the memory controller and then references this memory region to reinstate the remainder of the system.

Reliability–availability–serviceability

The importance customers place on the RAS characteristics of server-class computers compels server manufacturers to attain the highest cost-effective RAS. Main-memory compression adds a new facet to this endeavor [7], with the primary goal of detecting any data corruption within the system. Toward that end, MXT includes many RAS-specific features with appropriate logging and programmable interrupt control:

- Sector translation table entry parity checking.
- Sector free-list parity checking.
- Sector out-of-range checking.
- Sectored memory-overrun detection.
- Sectors-used threshold detection (2).
- Compressor/decompressor validity checking.
- Compressed-memory CRC protection.

Since the compression and decompression functions effectively encode and decode the system data, any malfunction during these processes can produce output that is seemingly correct, yet corrupted. Further, the hardware function implementation requires a prodigious quantity (of the order of one million) of logic gates. Although special fault-detection mechanisms are incorporated within the compression/decompression hardware, they cannot provide complete fault coverage. Consequently, there is a reasonable probability that a logic-upset-induced data corruption may survive undetected. Therefore, we needed an improved method of data integrity protection to minimize the potential for corrupted data to persist in the system without detection.

We employed a standard 32-bit cyclic redundancy code (CRC) computation over the uncompressed data block as it streams into the compressor. When the compression is complete, and the data is to be stored in the compressed format (i.e., if the data is compressible, such that a spatial advantage exists over storing the data in the uncompressed format), the check code is appended to the end of the compressed data block, and the associated block size is increased by four bytes. Information that is stored in the

uncompressed format gains little benefit from the CRC protection because it bypasses the compressor and decompressor functions, and hence is not covered by the CRC protection. Servicing a compressed-memory read request results in the decompression of the compressed block and concurrent recomputation of the CRC over the uncompressed data stream from the decompressor. Upon completion of the decompression, the appended CRC is compared to the recomputed CRC. When the two are not equal, an uncorrectable error is signaled within the system to alert the operating system to the event. Retrying the errant request may avoid the uncorrectable error condition when the error is caused by a transient upset within the decompressor. However, an error condition is most likely caused by the compressor, and retrying the request will have no beneficial effect in such a case. Therefore, there is little compelling benefit for supporting retry behavior.

Commodity duplex memory

Some system applications demand levels of RAS beyond that typically available in commercial systems. The fault-tolerant systems that are available are often cost-prohibitive for applications, except for small specific-application niches. The extra cost for such systems can be attributed to the cost of replicating hardware to provide redundancy for continued operation in the presence of a hardware fault. An MXT architecture extension permits fault-tolerant "duplex" or mirrored memory systems to be constructed without the extra cost of duplicating the memory devices, since the main-memory compression more than compensates for the redundant storage. These systems facilitate tolerating and repairing faults within the main memory without interruption of application or operating system software operation.

The architecture extends the memory controller to support not only a single unified memory, but a dual-redundant duplex memory capable of operating in the presence of a hardware failure or maintenance outage. The system structure, shown in **Figure 8**, incorporates the necessary electrical isolation/rebuffer mechanism and independent memory card power supplies to permit operation as a conventional, or optionally as a duplex, memory machine. Data errors are detected using the conventional memory error-detection and -correction hardware. When configured for duplex memory operation, identical content is maintained within each memory bank, such that any uncorrectable data error detected upon read access to a given bank may be reread from the other bank with the intent of receiving data without error. After a memory bank is identified as faulty, the memory controller can preclude further read access to the bank, permitting replacement without interruption to the application or operating system software.

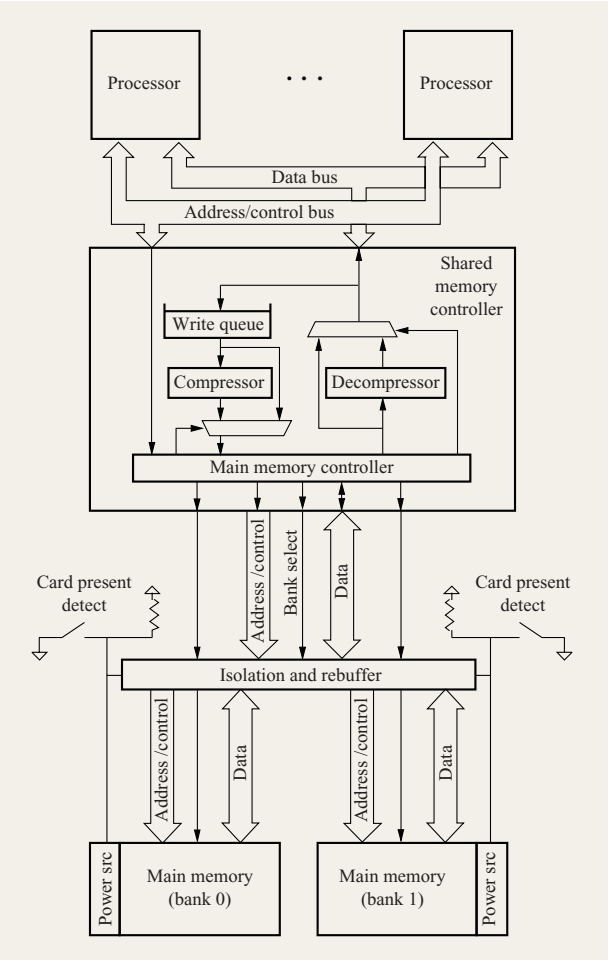

Figure 8

Mirrored memory block diagram.

Typically, each bank comprises a field-replaceable memory circuit card, which contains a quantity of SDRAM packaged as dual in-line memory modules (DIMMs). All activity may be configured to occur concurrently in order to maintain "lock-step" synchronization between the two banks. Although memory read accesses always occur at both banks, the electrical isolation mechanism, shown in **Figure 9**, provides a means for the memory controller to receive data selectively from only one of the two memory banks, known as the *primary* bank, whereas the "ignored" bank is known as the *backup* bank. However, write accesses always occur at both banks.

The memory controller functions are designed with special consideration for duplex memory operation. A memory "scrub" controller mode, which immediately reads and then writes back every location in memory, provides a means for copying the content from the primary bank to the backup memory bank. Further, the scrub

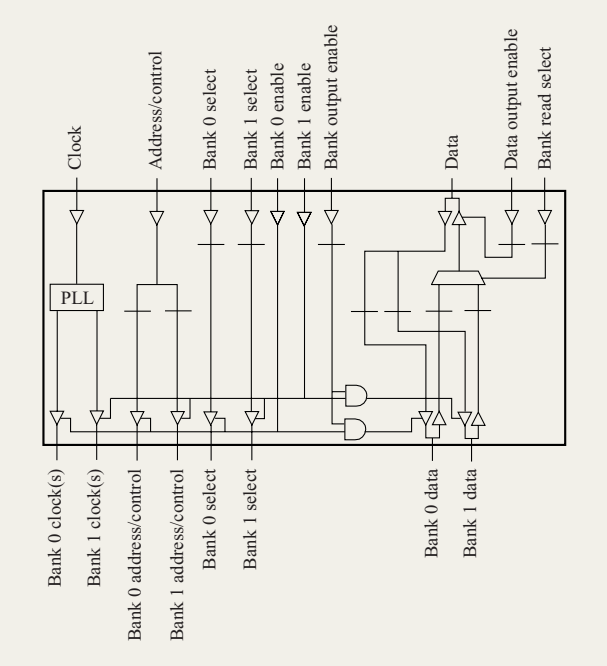

Figure 9

Isolation and rebuffer block diagram.

controller alternates normal memory-scrub read access between the two banks to ensure that the backup bank has not accumulated any content errors. Finally, the memory controller can be configured to "fail-over" to the backup bank from the primary bank upon detection of an uncorrectable error from the ECC in the data-read path. The fail-over process involves swapping the bank configuration (backup bank becomes primary bank and vice versa), and reissuing the read operation to receive the reply data from the new primary bank. Several "duplex" modes exist, permitting manual selection, automatic fail-over trip event, or automatic fail-over toggle event.

The memory controller may be configured to operate in any one of six modes (defined below) for utilizing the system memory banks. While all modes are user-selectable, modes 4–6 permit control hardware modification as well.

1. *Normal operation* Either one or both of the memory cards can be addressed and accessed independently. Bank "0" contains the low-order addressed memory, and bank "1" contains the high-order addressed memory. For a given read access, the *bank select* signal state corresponds to the addressed bank.
2. *Bank "1" mirrors bank "0"* Read and write accesses are presented to both cards simultaneously, but read data is selected from bank "0" via the bank select signal. This mode provides a means to logically ignore bank "1"; bank "1" is thus permitted to be either in or out of the system in support of repair and/or replacement.
3. *Bank "0" mirrors bank "1"* Read and write accesses are presented to both cards simultaneously, but read data is selected from bank "1" via the bank select signal. This mode provides a means to logically ignore bank "0"; bank "0" is thus permitted to be either in or out of the system in support of repair and/or replacement.
4. *Bank "1" mirrors bank "0" with automatic fail-over to mode 3* Read and write accesses are presented to both cards simultaneously, but read data is selected from bank "0" via the bank select signal. Upon detecting an uncorrectable error (UE) within a read reply, the memory controller will reclassify the error as a correctable error, reconfigure the memory bank operation to mode 3, and retry the memory read access. Any UE detection during a scrub operation to bank "1" results in automatic reconfiguration to mode 2.
5. *Bank "1" mirrors bank "0" with automatic fail-over to mode 6*.
6. *Bank "0" mirrors bank "1" with automatic fail-over to mode 5.*
   Modes 5 and 6 permit the memory system to tolerate multiple faults across the two banks, as long as the faults do not exist at the same addresses. This is accomplished by toggling between modes 5 and 6 each time a UE is detected within a read reply. Upon detecting a UE, the memory controller will reclassify the error as a correctable error, reconfigure the memory bank operation to the alternate mode, and retry the memory read access. These modes serve to extend the mean time to repair (MTTR) of a system memory fault, but at the expense of losing some on-line maintainability when both cards have accumulated a UE. A replacement memory card can be initialized only with the content of the existing card, which in this case is contaminated with a UE.

Another register is used to provide user control of new and special functions of the memory scrub hardware within the memory controller. These functions are unique and necessary to the operation of duplex memory operation:

1. *Scrub immediate* Read and write successive blocks over the entire memory range without regard to a slow-paced "background" interval. When used in conjunction with mode 2 or mode 3, this function provides a means to read all of the content of the primary memory bank, validate the data integrity through EDC circuits, and rewrite the data back to both banks, for the purpose of

reinitializing the content of the backup bank from that of the primary bank, thus permitting a newly installed "empty" memory to be initialized while the system is in continuous use.

2. *Scrub background* Read (and write-only on correctable-error) successive blocks over the entire memory range with regard to a slow-paced "background" interval. This typical scrub operation is enhanced to support duplex operation by alternating entire memory scrubs between primary bank and backup banks when modes 4–6 are selected. This prevents the backup bank from accumulating "soft errors," since data is never actually received from the backup bank during normal read references.

Operating system software

All commercial computer operating system (OS) software environments manage the hardware memory as a shared resource to multiple processes. In cases in which the memory resource becomes limited (i.e., processes request more memory than is physically available within the machine), the OS can take steps for continued system operation. Typically, the OS migrates under-utilized memory pages (4KB) to disk, and then reallocates the memory to the requesting processes. In this manner, the main memory is used like a cache that is backed by a large disk-based storage. This scheme works quite well because the absolute amount of memory is known to the OS. This algorithm applies to MXT-based systems as well.

Although current "shrink wrap" OS software can be used on an MXT machine, the software cannot yet distinguish an MXT-based machine from a conventional memory hardware environment. This poses a problem, since the amount of memory "known" to the OS is twice what actually exists within an MXT machine. Further, the OS is not aware of the notion of compression ratio. Therefore, the OS cannot detect conditions in which the physical memory may be over-utilized (i.e., there are too few unused or free sectors left in the sectored memory), and therefore may not invoke the paging management software to handle the situation, possibly leading to a system failure. This condition can occur when the OS has fully allocated the available memory and the overall compression ratio has fallen below 2:1.

Fortunately, minor changes in the OS kernel virtual memory manager are sufficient to make the OS "MXT-aware" [8]. Further, the same objective can also be accomplished outside the kernel, for example in a device driver or service, albeit less efficiently.

Performance

MXT compression performance fundamentally ranges between 1:0.98 (1:1) and 64:1, including translation table memory overhead. **Figure 10** shows a representative

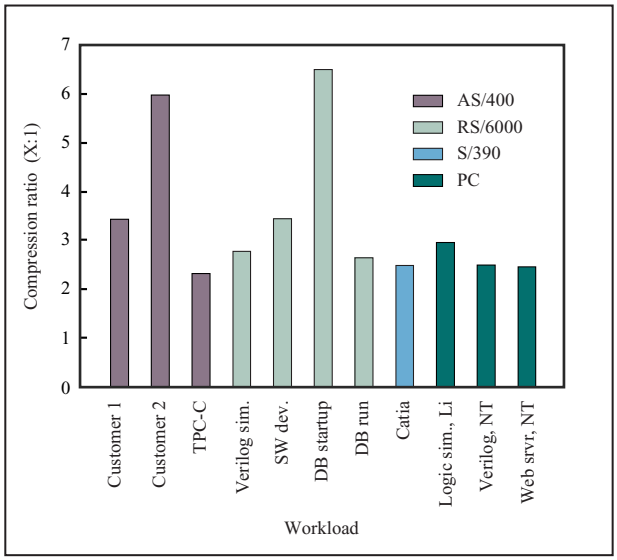

Figure 10

Memory compressibility.

sampling of the many memory-content compressibility measurements we have taken from several types of machines. We can take measurements by direct measurement on an MXT machine, by indirect measurement via a monitor program running on a non-MXT machine, and of course by post-analysis of memory dumps. Compressibility drops below 2:1 only in the rare case in which the majority of the system memory contains random or precompressed data.

We have observed that the compression ratio of a given machine tends to remain relatively constant throughout the operation of the application set. For example, monitoring the IBM Internet on-line-ordering web server[2] over a period of ten hours indicated a compression ratio of 2.15:1 ± 1%. Further, it can be seen in **Figure 11** that the distribution of compressibility is normal. Each bar of the histogram represents the degree of compressibility, with the rightmost bar incompressible (1:1) and the leftmost bar maximally compressed (64:1). The lower curve represents the degree of change in compressibility over the measurement period.

MXT system performance evaluation can be considered from two primary perspectives: intrinsic performance, such as that measured on any conventional system, and cost/performance in memory-starved applications. Much has been written about the performance benefit additional memory can provide for memory-intensive applications. As one might expect, this is where MXT systems really stand out, so much so that we typically see customers

[2] Specific shadow servers 9q, 9w for website *http://www.pc.ibm.com/ibm_us/*.

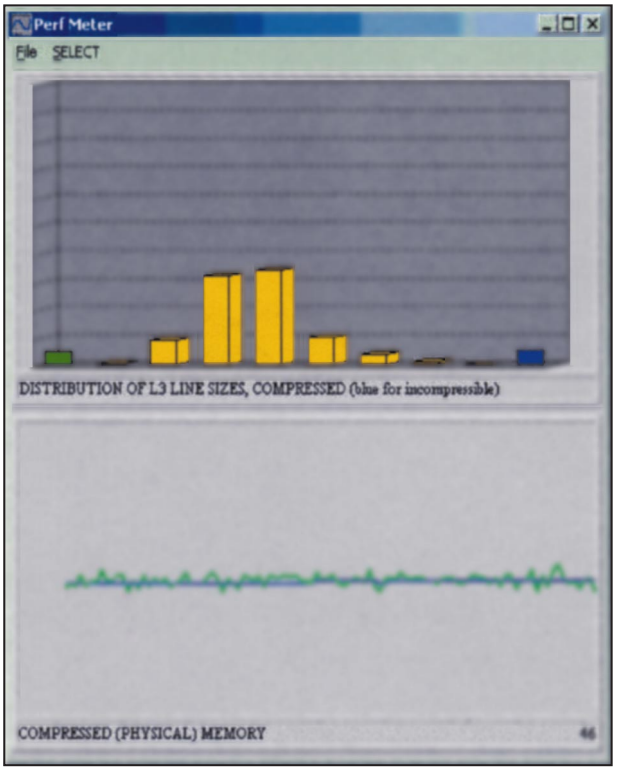

Figure 11

IBM website compression distribution.

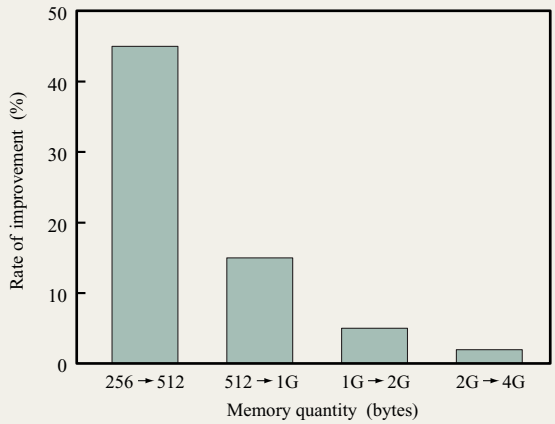

Figure 12

Dependence of SPECweb99 performance (number of simultaneous connections) on available system memory.

experiencing 50–100% improvement in system throughput. For example, one customer operating a computer farm with several thousand dual-processor servers, each containing 1 GB of memory, was able to run one job per unit time on each machine. When an equivalent (dual-processor and 1GB memory) MXT-based machine was used in the environment, two jobs could be run concurrently over the same period of time, because the 1 GB was effectively doubled to 2 GB through MXT expansion. Similar memory-dependent performance is observed with the behavior of the well-known SPECweb99** benchmark. For this case (**Figure 12**), increasing memory from 256 MB to 512 MB yields a 45% performance improvement (but beyond 512 MB the benefit diminishes).

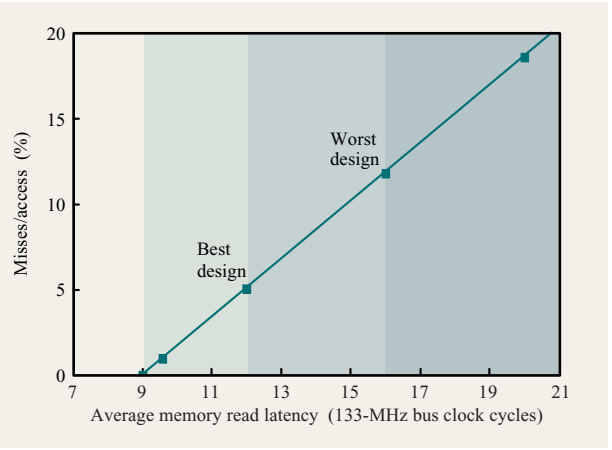

Figure 13

Cache miss rate vs. system memory performance.

We began this project with the primary goal of gaining the benefit of doubling the system memory at a negligible cost, but without degrading system performance. To that end, MXT-based memory controller performance is based on the intrinsic hardware implementation reaction time, as well as the shared-cache hit rate. When an MXT memory controller employs an independent shared cache, such as that used in the Pinnacle chip [3], the average read latency can be plotted as a function of the cache miss rate, as shown in **Figure 13**. The region to the left of the "best design" represents the average read latency for the MXT-based memory controller. The region between the "best design" and "worst design" points reflects the average read latency for contemporary conventional "cacheless" memory controllers available in the marketplace.

The shared-cache hit rate is application-dependent, but we typically measure the cache hit rate at roughly 98% on most applications. However, the cache hit rate for large database applications such as TPC-C**, SAP**, and particularly Lotus Notes** can range as low as 94%, as

measured by quad-processor trace-driven performance models. These applications tend to reference some database records infrequently, resulting in a prefetch advantage with the long cache line but little reuse of the data within the line.

Our comparison of MXT system performance with that of a high-performance contemporary system showed the two systems having essentially equivalent (within one point) performance for the SPECint2000** benchmark. Both machines were IBM 1U commercial servers with 512 MB and Intel 733-MHz PIII processors, executing program code from the same disk drive. The two systems differ only in the type of memory controller used. While the MXT system used the ServerWorks Pinnacle chip, the other system used the ServerWorks CNB30LE chip.

MXT provides a system cost leverage not seen since the invention of DRAM. **Figure 14** illustrates the degree of this leverage with a case in point. The graph shows how the cost/performance metric for a family of conventional machines (dashed curve) is dramatically improved when the benefits of MXT are factored in (solid curve). For a representative product, we configured a ProLiant DL360 commercial server on the COMPAQ Computer Corporation Internet website[3] for retail equipment sales. This server was priced at $9759. Using the same site to configure a hypothetical MXT equivalent system, with half the memory and an estimated $150 MXT cost premium, yielded nearly 30% savings at $6904. Viewed another way, a hypothetically "more capable" MXT system can be configured at a cost ($9702) commensurate with that of the reference machine. Either way, an MXT system cost/performance metric compares quite favorably with that of any conventional system.

MXT is a logical step in the evolution of compression technologies, and is a proven technology that empowers customers to efficiently utilize their memory investment. Information technology professionals can routinely achieve significant savings on systems ranging from high-density servers to large memory enterprise servers. We expect MXT to expand its presence into other processor memory controllers and memory-intensive system applications, including disk storage controllers, laptop computers, and information appliances.

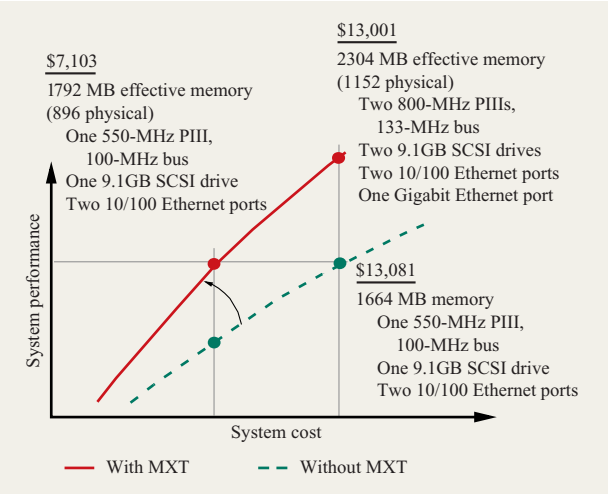

Figure 14

System cost comparison.

Acknowledgments

The MXT architecture involved contributions from several other individuals. We especially thank Philip Heidelberger, Vittorio Castelli, and Caroline Benveniste for their work on the compression efficiency and performance analysis; Dan Poff, Rob Saccone, and Bulent Abali for operating system and performance measurement work; and Michel Hack for his work on storage data structures.

[3] Referenced on November 29, 2000, at website *http://www5.compaq.com/products/servers/platforms*.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Intel Corporation, Standard Performance Evaluation Corporation, Transaction Processing Performance Council, SAP AG, or Lotus Development Corporation.

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, September 22, 1998.
2. M. Kjelso, M. Gooch, and S. Jones, "Design and Performance of a Main Memory Hardware Data Compressor," *Proceedings of the 22nd EUROMICRO Conference*, IEEE, 1996, pp. 423–430.
3. R. B. Tremaine, T. B. Smith, M. Wazlowski, D. Har, K. Mak, and S. Arramreddy, "Pinnacle: IBM MXT in a Memory Controller Chip," *IEEE Micro* **22,** No. 2, 56–68 (March/April 2001).
4. M. Abbott, D. Har, L. Herger, M. Kauffmann, K. Mak, J. Murdock, C. Schulz, T. B. Smith, B. Tremaine, D. Yeh, and L. Wong, "Durable Memory RS/6000 System Design," *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing*, IEEE, 1994, pp. 414–423.
5. P. A. Franaszek and J. Robinson, "On Internal Organizations in Compressed Random Access Memories," *Research Report RC-21146*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, April 1998.
6. P. A. 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.
7. C. L. Chen, D. Har, K. Mak, C. Schulz, B. Tremaine, and M. Wazlowski, "Reliability–Availability–Serviceability of a Compressed Memory System," *Proceedings of the*

International Symposium on Dependable Systems and Networks, IEEE, 2000, pp. 163–168.
8. B. Abali and H. Franke, "Operating System Support for Fast Hardware Compression of Main Memory Contents," *Proceedings of the Memory Wall Workshop*, held in conjunction with the 27th International Symposium on Computer Architecture (ISCA-2000), Vancouver, BC, June 2000.

Received November 1, 2000; accepted for publication January 26, 2001

R. Brett Tremaine *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (afton@us.ibm.com).* Mr. Tremaine is a Senior Technical Staff Member at the IBM Thomas J. Watson Research Center, where he is responsible for commercial server and memory hierarchy architecture, design, and ASIC implementation. Before joining the Research Division in 1989, he had worked for the IBM Federal Systems Division in Owego, New York, since 1982. He has led several server architecture and ASIC design projects, many with interdivisional relationships, and has received two IBM Outstanding Technical Achievement Awards and several division awards for his contributions. Mr. Tremaine received an M.S. degree in computer engineering from Syracuse University in 1988, and a B.S. degree in electrical engineering from Michigan Technological University in 1982. He has eleven patents pending and several publications, and he is a member of the IEEE.

Peter A. Franaszek *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (paf@us.ibm.com).* Dr. Franaszek received the Ph.D. degree in electrical engineering from Princeton University in 1965. From 1965 to 1968, he was employed by Bell Laboratories. He joined the IBM Research Division in 1968. During the academic year 1973–1974, he was on sabbatical leave at Stanford University as Consulting Associate Professor of Computer Science and Electrical Engineering. He is currently Manager of Systems Theory and Analysis. His interests are in the general area of information representation and management, and computer system organization. Dr. Franaszek has received two IBM Corporate Awards for his work on codes for magnetic recording, an IBM Corporate Patent Portfolio award for his contribution to the ESCON architecture, and Outstanding Innovation Awards for fragmentation-reduction algorithms, network theory, concurrency-control algorithms, run-length-limited codes, and the code used in ESCON, Fiber Channel, and Gigabit Ethernet. He is a member of the IBM Academy of Technology and a Master Inventor. He is a Fellow of the IEEE, and received the 1989 Emmanuel R. Piore Award from the IEEE for his contributions to the theory and practice of constrained channel coding in digital recording. Dr. Franaszek holds thirty-six patents and has published more than forty technical papers.

John T. Robinson *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (robnson@us.ibm.com; http://www.research.ibm.com/people/r/robnson/).* Dr. Robinson received the B.S. degree in mathematics from Stanford University in 1974, and the Ph.D. degree in computer science from Carnegie Mellon University in 1982. Since 1981, he has been with the IBM Thomas J. Watson Research Center, Yorktown Heights, New York. His research interests include database systems, operating systems, parallel and distributed processing, design and analysis of algorithms, and hardware design and verification. He is a member of the ACM and the IEEE Computer Society.

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.

T. Basil Smith *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (tbsmith@us.ibm.com).* Since 1986 Dr. Smith has been a Research Staff Member at the IBM Thomas J. Watson Research Center, where he is now 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. He has received both 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, Texas, and at the Charles Stark Draper Laboratory in Cambridge, Massachusetts. Dr. Smith holds more than 20 patents in computer architecture and reliable machine design. He 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 City in June 2000.

Michael E. Wazlowski *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (mew@us.ibm.com).* Since 1995 Dr. Wazlowski has been a Research Staff Member at the IBM Thomas J. Watson Research Center, where he is responsible for high-performance memory system architecture and design. He received a B.S. degree in computer engineering from the University of Massachusetts at Amherst in 1990 and M.S. and Ph.D. degrees in electrical sciences from Brown University in 1992 and 1996, respectively. He is a co-inventor on several MXT patents. Dr. Wazlowski's research interests include computer architecture, memory systems, and ASIC design; he is currently leading the verification effort for a 1.5-million-gate ASIC. He is a member of the IEEE.

P. Maurice Bland *IBM Server Group, 3039 Cornwallis Road, Research Triangle Park, North Carolina 27709 (pmbland@us.ibm.com)*. Mr. Bland is a Senior Technical Staff Member in the eServer xSeries system development organization, where he is responsible for high-end Intel-based server system design and architecture. Since joining IBM in 1979, he has designed many PC-based products, ranging from laptops to servers. Mr. Bland has reached an 11th invention achievement plateau, having received more than 25 patents in I/O bus bridging, systems power management, memory subsystem design, and fault-tolerant system design. He received a B.S. degree in electrical engineering from the University of Kentucky in 1978.