IBM Memory eXpansion Technology (MXT)

R. Brett Tremaine[γ], T. Basil Smith, Mike Wazlowski

IBM T. J. Watson Research Center
P.O. Box 218
Yorktown, NY 10598

[γ]Email: afton@us.ibm.com
Phone: 914-945-2710
FAX: 914-945-2141

Abstract

Several state-of-the-art technologies are leveraged to establish an architecture for a low-cost and high performance memory controller and memory system that more than doubles the effective size of the installed main memory without significant added cost. This unique 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. Novel high-speed and 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 - 6:1, as measured in "real-world" system applications.

1.0 Introduction

Memory costs dominate both large memory servers and expansive compute server environments, like those employed in today's "data centers" and "compute 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 tradeoff against compute 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 [7][3], system main memory compression has not been exploited to its potential. IBM's Memory eXpansion Technology (MXT) addresses the system memory cost issue head-on 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 PentiumIII/Xeon Bus compatible, low cost single chip memory controller (north bridge) [8]. This unique chip is the first commercially available memory controller to employ real-time main memory content compression at a performance competitive with the market's best products.

2.0 Architecture

Conventional "commodity" computer systems typically share a common architecture; where a collection of processors are connected to a common SDRAM based main memory through a memory controller chip set. MXT incorporates 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 to overall system performance.

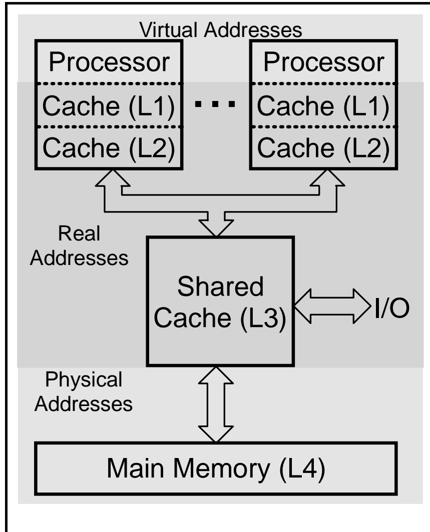


Figure 1: System Memory Hierarchy.

The shared cache, coupled with the recent advent of high density 0.25 micron 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 main memory cost, 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 IO memory references are directed to the cache controller, resulting in cache directory lookup to determine if the address is contained within the cache or not. 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 back the old cache line to the write back buffer (wtq) in those cases where 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 commences reading 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 7 cycle advance notification of when the requested 32B data will be in the critical word buffer (cw). This permits the processor bus controller to arbitrate for a deferred read reply to the processor data bus, and deliver data without delay.

Cache write back activity is processed in parallel with read activity. Once an entire cache line is queued in the write back buffer (wtq), the compression commences and runs uninterrupted until complete, 256 cycles later. Then the memory controller stores the compressed data, when a spatial advantage exists, otherwise the memory controller stores the write back data directly from the write back buffer. In either case, the memory controller must first read the translation table entry for the write back address 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.

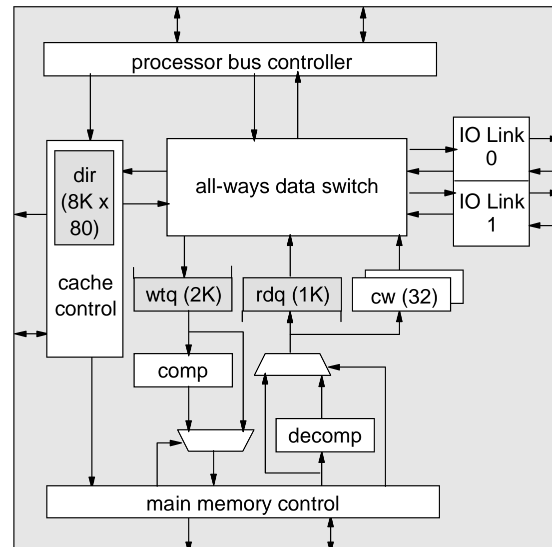


Figure 2: Typical control chip block diagram.

2.1 Shared Cache Subsystem

The shared cache provides low latency processor and IO subsystem access to frequently accessed uncompressed data. The data/code/IO 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*, where each line is 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 that 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 one of three primary architectural structures, where performance can be traded off with the cost of implementation:

1. The independent cache array scheme [1], provides the greatest performance, but at the highest cost. The large independent data cache memory is implemented in 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 largely limited by the size of the cache directory that can be fit on the die. For example a single chip memory controller implemented in 0.25 micron CMOS, can support a 32MB cache, whereas in 0.18 micron a 64MB cache can be supported, assuming a 12mm die. The cost for this type of implementation ranges between $50-$60. However, the performance is maximized as 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 [7], 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 cost can range $0 - $30, and is most advantaged when storing the cache directory 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 quantity 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 quantity of uncompressed lines. Only the most recently used *n* lines are selected to comprise 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, as the 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 direct 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.

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 any one or more cache line segments. When a cache line is evicted, only "referenced" cache line segments, verses 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 write back 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 line to be written back, reloaded, and referenced simultaneously during cache line replacement.

During replacement, a state vector is maintained to indicate *old, new*, or *invalid* state for each of the thirty-two, 32B sub-cache lines within the physical line. As 32B sub-cache lines are invalidated or moved from the cache to the write back buffer, the sectors are marked *invalid*, indicating that the associated new sub-cache line may be written into the cache. Each time a new sub-cache line is loaded, the associated state is marked *new*, indicating that processor/IO

access is permitted to the new cache line address. Further, processor/IO accesses to the old cache address are also permitted, when the associated sub-cache lines are marked *old*. Cache lines are always optimally fetched and filled, such that the sub-cache line write back follows the same sub-cache line order to maximize the amount of valid cache line at all times.

The cache supports at least two concurrent cache line replacements. Two independent 1KB write back 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 write back 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 32B granule 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.

2.2 Main Memory Subsystem

The main memory subsystem stores and retrieves cache lines in response to shared cache write back (write) and line fill (read) requests. Data is stored within the main memory array is comprised of industry standard SDRAM DIMM's. 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 DIMM's directly connected to the memory controller chip(s) without any "glue" logic. Whereas 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 32B first and 256B address wrapped as shown in Figure 3.

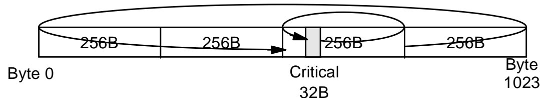
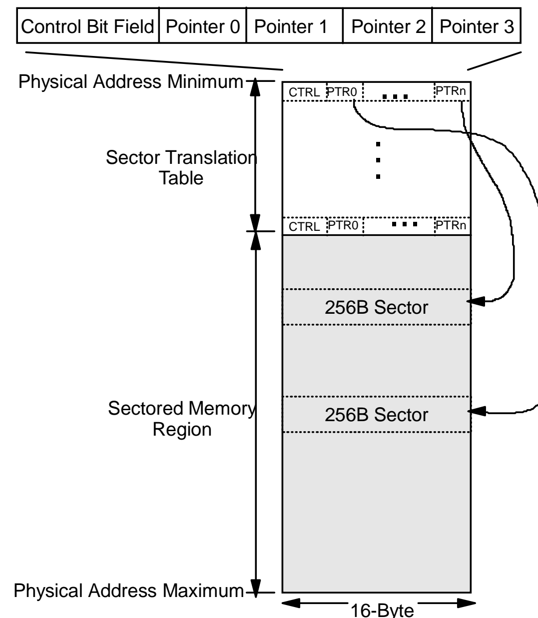


Figure 3: Critical word fetch order.

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 dynamically allocating storage for the variable size 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 is comprised of two primary data structures; the *sector translation table* (STT) and the *sectored memory* as shown in Figure 5. The STT consists of an array of 16B entries, where 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



[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 quantity and type of DIMM's installed in a system. When compression is enabled, the BIOS doubles this value to indicate a real address space twice as large as is populated with DIMM's.

Figure 4: Memory organization.

When a 1KB data block is compressible to less than 121 bits, then 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 1-4, 256B sectors, with the sector pointers (PTR) contained within the STT entry, as shown in the table below. For the case where the data block is uncompressed, four sectors are used and the STT entry control field indicates the "uncompressed" attribute. In cases where 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 2-way sharing scheme typically improves the overall compression efficiency by 15%, nearly all the potential gain attainable from combining fragments by any degree of sharing [6].

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 6, each node of the free list contains pointers to sixty-three 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, effectively the free-list requires no additional storage.

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.

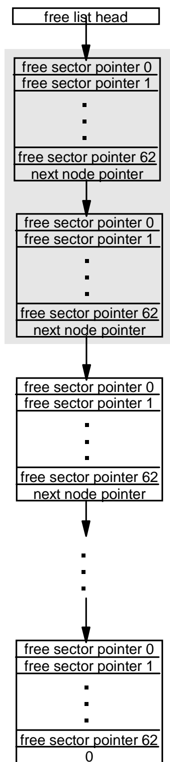


Figure 5: Free list.

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-256MB range, 32KB aligned. These 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 as data is directly addressable without the intermediate step of referencing a STT entry. And of course, the data is fetched with the requested 32B first, as is always the case for uncompressed data.

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 on to the free list.

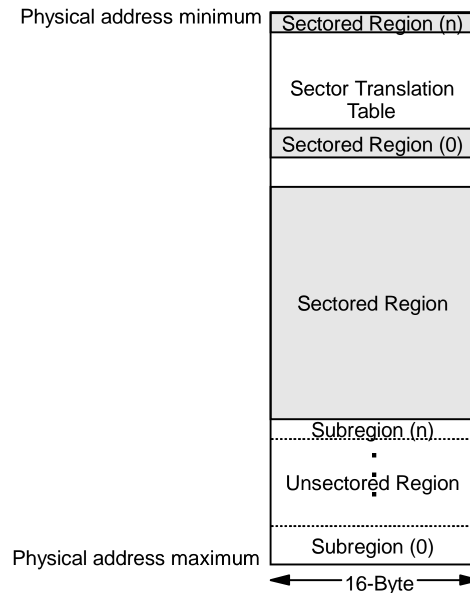


Figure 6: Unsectored memory organization.

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, and de-allocate

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>

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 when a cache line is 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.

2.3 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 0.1-3.0 micro-seconds, 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 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 memory usage optimization algorithms.

Page operations are initiated atomically to the memory controller, i.e., the controller will not accept a new page request until any pending request is completed. This insures 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 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 memory hierarchy state. 3) Swap STT entry with complement page STT entry.

0110 – Swap with complement page (defined in *Complement Page* Register). 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 STT entry with 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 STT entry with 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

2.4 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 loss-less. Although a plethora of compression algorithms exist, none met our architectural criteria. We chose to leverage the recently available high density (0.25 micron) CMOS ASIC technology by implementing a gate intensive, parallelized derivative [2] 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 essentially equivalent compression efficiency as the sequential LZ77 method [2].

Typically four compression engines are employed, each operating on 256B (¼ of the 1KB uncompressed data block), at the rate of 1B/cycle, yielding a 4B/cycle aggregate compression rate. Referring to Figure 7, each engine contains a history buffer or *dictionary* consisting of a 255-byte Content Addressable Memory (CAM) that functions as a shift register. Attached to each dictionary are four 255-byte (4080) comparators for locating the incoming reference byte within the entire dictionary structure. Each clock cycle, one byte

from each 256B source data block (read from the shared cache write back 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*.

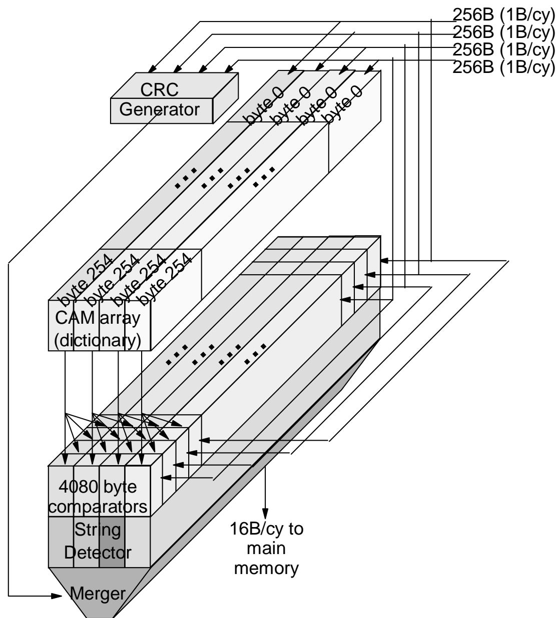


Figure 7: Compressor block diagram.

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 the table below, that the 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 when the accumulated aggregate compressed output exceeds 1KB (or a programmed threshold), causing compression to be aborted and the uncompressed data block stored in memory.

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 starting nearest the beginning of the dictionary for multiple strings with the same length, is selected. The length field ranges from 2-12 bits to encode the number of bytes in the *working string*, using a Huffman coding scheme. The position field ranges from 2-10 bits to encode the starting address of the *reference string*. Merge logic is responsible for packing the variable length bits 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 write back 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 is comprised of four engines the each decoding the encoded compressed data block. Each engine can produce 2B/cycle, yielding an aggregate 8B/cy when 1X clocked or 16B/cycle when 2X clocked, as occurs in Pinnacle.

2.5 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 a NV main memory implementation, in lieu of copying information back to magnetic disk during power outages. A low-power "sleep" state, where only the SDRAM memory remains powered by a back-up 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 back-up 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, as 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 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 on impending power loss, software can store the entire system state into 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.

3.0 Reliability-Availability-Servicability

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 [4], with the primary goal of detecting any data corruption within the system. Toward that end, MXT includes many RAS specific features (delineated below), 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 the processes can produce seemingly correct, yet corrupted output. Further, the hardware function implementation requires a prodigious quantity of (order 1 million) logic gates. Thus, the probability for a logic upset induced data corruption that goes undetected is significant. Although special fault detection mechanisms within the compression/decompression hardware have been incorporated, they cannot provide complete fault coverage. 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., 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 4 bytes. Information that is stored in the uncompressed format gains little benefit from the CRC protection as 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 re-computed CRC. When the two are not equal, an uncorrectable error is signaled within the system to alert the operating system to the event.

3.1 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 outside of 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, as 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.

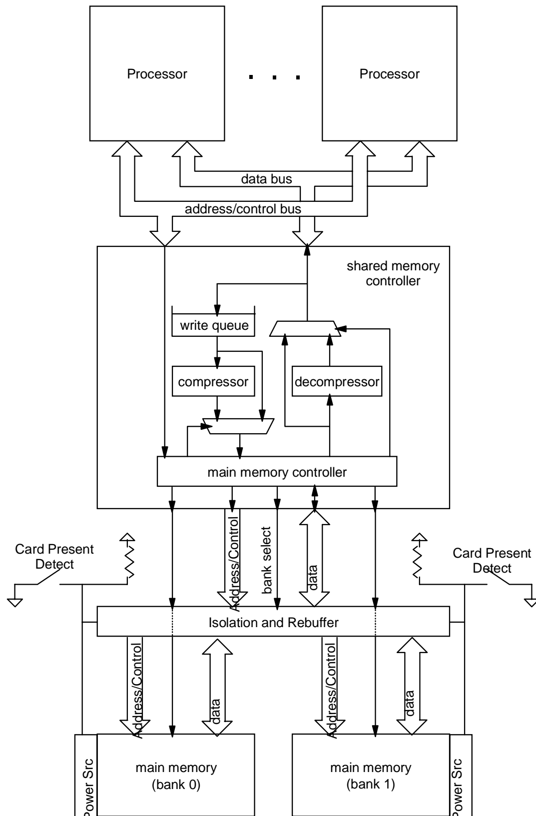


Figure 8: Mirrored memory block diagram.

Typically, each bank is comprised of a field replaceable memory circuit card, which contains a quantity of SDRAM packaged as dual inline memory modules (DIMM's). All activity may be configured to occur concurrently 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 selectively receive data from only one of the two memory banks, known as the *primary* bank, whereas the "ignored" bank is known as the *back-up* bank. However, write accesses always occur at both banks.

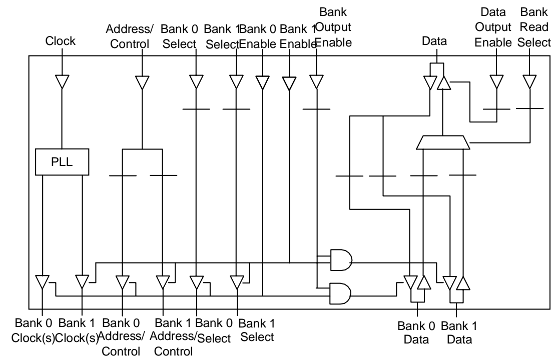


Figure 9: Isolation and rebuffer block diagram.

The memory controller functions are designed with special consideration for duplex memory operation, including: A memory scrub controller mode to immediately scrub the entire memory address space by reading and then writing the content back for the purpose of initializing an *back-up* memory bank with the content from the *primary* bank. Further, the scrub controller alternates normal memory scrub read access between the two banks to insure the *back-up* bank has not accumulated any content errors. Lastly, the memory controller can be configured to fail-over to the *back-up* 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 (*back-up* 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 too.

1. **Normal operation:** Either one or both memory cards are independently addressed and accessed. 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 is 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", thus permitting bank "1" 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 is to both cards simultaneously, but read data is selected from bank "1", via *bank select* signal. This mode provides a means to logically ignore bank "0", thus permitting bank "0" 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 the mode 3:** Retry read after an uncorrectable error (UE) is detected during a read reply. Upon detecting a UE, the memory controller will reclassify the error as a correctable error, and retry the memory read access and any future access with the alternate bank.
5. **Bank "1" mirrors bank "0" with automatic fail-over to the mode 6.**
6. **Bank "0" mirrors bank "1" with automatic fail-over to the 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 an uncorrectable error (UE) is detected during a read reply. Upon detecting a UE, the memory controller will reclassify the error as a correctable error, and retry the memory read access with the any future access with the alternate bank. These modes serve to extend the mean time to repair (MTTR) of a system memory fault, but at the expense of loosing some on-line maintainability when both cards have accumulated an UE. A replacement memory card can only be initialized with the existing card's content, 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, including:

1. **Scrub Immediate** - Scrub read and write successive blocks over the entire memory range without regard to a slow pace "background" interval. When used in conjunction with the aforementioned mode 2 or 3, this function provides a means to read all 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 *back-up* 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** - Scrub read (and write only on correctable error) successive blocks over the entire memory range with regard to a slow pace "back ground" interval. This typical scrub operation is enhanced to support duplex operation by alternating entire memory scrubs between *primary* bank and *back-up* banks when modes 4-6 are selected. This prevents the *back-up* bank from accumulating "soft errors", since data is never actually received from the *back-up* bank during normal read references.

4.0 Operating System Software

All commercial computer operating system (OS) software environments manage the hardware memory as a shared resource to multiple processes. In cases where 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 underutilized 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 still 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, as 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 either. Therefore, the OS cannot detect conditions when 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" [5]. Further, the same objective can also be accomplished outside the kernel, for example in a device driver or service, albeit less efficiently.

5.0 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 sampling of the many memory content compressibility measurements we have taken from

several types of machines. We can take measurements by either, direct measurement on an MXT machine, indirect measurement via a monitor program running on a non-MXT machine, and of course post analysis of memory dumps. Compressibility only drops below 2:1 in the rare case where the majority of the system memory contains random or pre-compressed data.

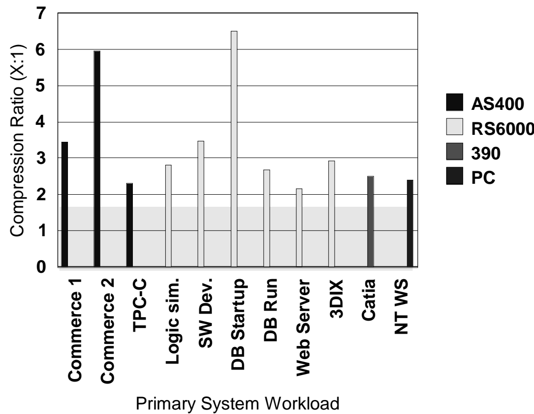


Figure 10: Memory compressibility.

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 online ordering web server[2] over a period of 10 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, where the right most bar is incompressible (1:1), and the left most bar is maximally compressed (64:1). The lower line represents the degree of change in compressibility over the measurement period.

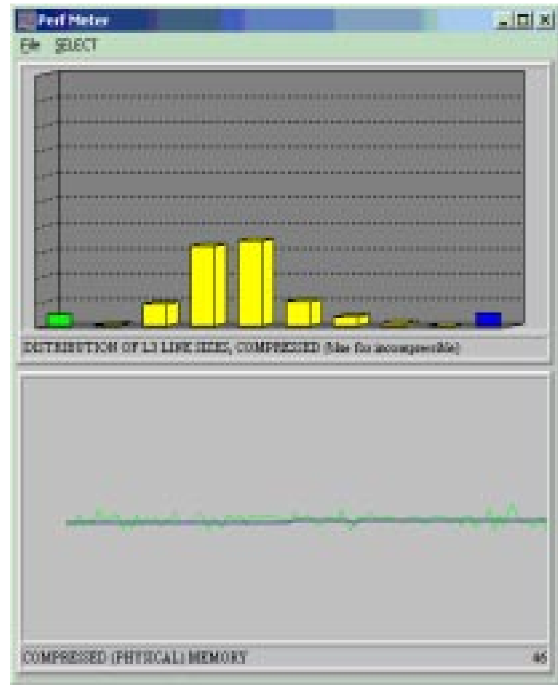


Figure 11: IBM web site compression distribution.

MXT system performance evaluation can be considered from two primary perspectives; one being the intrinsic performance like that measured on any conventional system, and the other being 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 experiencing 50%-100% improvement in system throughput. For example, one customer operating a compute farm with several thousand dual processor servers, each containing 1GB 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 1GB was effectively doubled to 2GB 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 256MB to 512MB yields a 45% performance improvement, beyond 512MB the benefit diminishes.

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

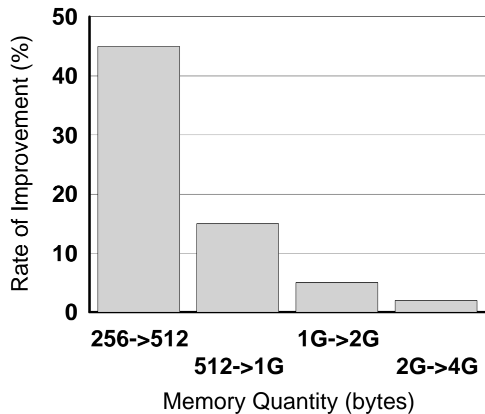


Figure 12: SPECweb99 performance (amount of simultaneous connections) dependence on available system memory.

We began this project with the primary goal of gaining the benefit of doubling the system memory at a negligible cost, but without degrading the 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, like that used in the Pinnacle chip [8], 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. Whereas the region between the "best design" and "worst design" points reflects the average read latency for contemporary conventional memory controllers available in the marketplace.

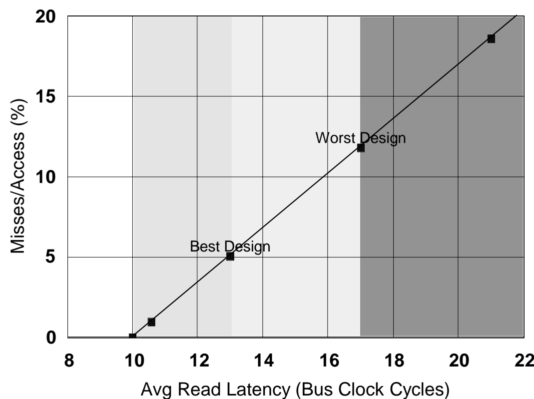
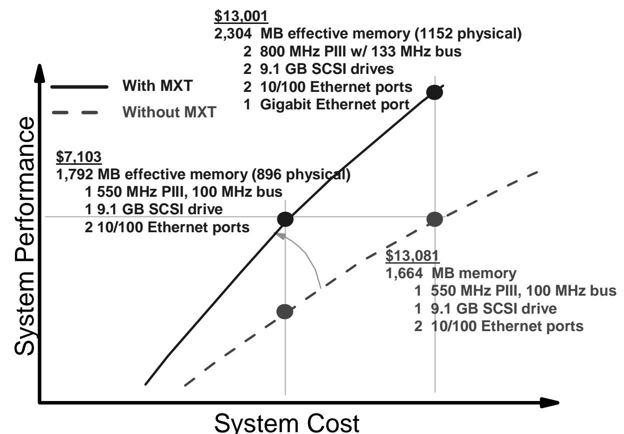


Figure 13: Cache miss rate vs. System memory performance.

The shared cache hit rate is application dependent, and as expected, we typically measure the cache hit rate at roughly 98% on most applications. However, the cache hit rate for large data base applications like 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, resulted with the two systems having essentially equivalent (within one point) performance for the SPECint2000 benchmark. Both machines were IBM 1U commercial servers with 512MB and Intel 733MHz 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 line) is dramatically improved when the effects of MXT are factored in (solid line). We configured a ProLiant DL360 commercial server on the Compaq Inc. Internet web site[3] for retail equipment sales. This server was priced at $13,081. Using the same site to configure a hypothetical MXT equivalent system with half the memory, yielded over 40% savings at $7,103. Viewed another way, a hypothetical MXT "better" system can be configured at the equivalent price to the reference machine. Either way, an MXT system cost-performance metric compares quite favorably with any conventional system.



[3] Referenced on September 11, 1999 at web site: http://www5.compaq.com/products/servers/platforms

Figure 14: System cost comparison.

MXT is a logical step in the pervasion of compression technologies, and is a proven technology that empowers customers to efficiently utilize their memory investment. Information Technology professionals can routinely save $1000's on systems ranging from high density servers to large memory enterprise servers. We expect MXT to expand its presence into other processor memory controllers, as well as other memory intensive system applications including, disk storage controllers, laptop computers, and information appliances.

Acknowledgments

The MXT architecture involved significant contributions from many other people. We especially thank: Peter Franaszek and John Robinson for their work on the compression approach; Dan Poff, Rob Saccone and Bulent Abali for operating system and performance measurement work; Michel Hack and Chuck Schulz for their work on the data storage and page operations.

References

[1] M. Abbott, D. Har, L. Herger, M. Kauffmann, K. Mak, J. Murdock, C. Schulz, T.B. Smith, B. Tremaine, D. Yeh, L. Wong *Durable Memory RS/6000 System Design*, Digest of Papers, The 24th Annual International Symposium on Fault-Tolerant Computing, Austin, Texas June 15-17, pp. 414-423, 1994.

[2] P. A. Franaszek, J. Robinson, J. Thomas, Parallel compression with cooperative dictionary construction. In *Proceedings DCC '96 Data Compression Conference*, pp. 200-209, IEEE, 1996

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

[4] C. L. Chen, D. Har, K. Mak, C. Schulz, B. Tremaine, M. Wazlowski, "Reliability - Availability - Serviceability of a Compressed Memory System," In *Proceedings of the International Symposium on Dependable Systems and Networks*, pp. 163-168, IEEE, 2000.

[5] B. Abali and H. Franke, "Operating System Support for Fast Hardware Compression of Main Memory Contents," in *Proc. Memory Wall Workshop*, held in conjunction with the 27th International Symposium on Computer Architecture (ISCA-2000). Vancouver BC, Canada, June 2000.

[6] P. A. Franaszek, J. Robinson, Design and Analysis of Internal Organizations for Compressed Random Access Memories, Research Report RC 21146(94535)20OCT1998, IBM Research, March 1992.

[7] Hovis, et all "Compression Architecture for system memory application", United States Patent US5812817, issued September 22, 1998.

[8] S. Arramreddy, D. Har, K. Mak, T.B. Smith, R.B. Tremaine, and M. Wazlowski, "IBM Memory eXpansion Technology (MXT) Debuts in a ServerWorks Northbridge", submitted for publication in IEEEmicro, October 13, 2000.