RC22320 (W0201-119) January 28, 2002
Computer Science

IBM Research Report

Achieving High Performance Network and Disk I/O with the Memory Xpansion Technology

Jiuxing Liu*, Mohammad Banikazemi, Dhabaleswar K. Panda*
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

*Dept. of Computer and Information Science
The Ohio State University
Columbus, OH 43210

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. , payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Achieving High Performance Network and Disk I/O with the Memory Xpansion Technology

Jiuxing Liu[†1] Mohammad Banikazemi[‡2] Dhabaleswar K. Panda[†]

[†]Dept. of Computer and Information Science
The Ohio State University
Columbus, OH 43210
email: {liuj, panda}@cis.ohio-state.edu

[‡]System Design & Performance
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
email: mb@us.ibm.com

Abstract

The Memory Xpansion Technology (MXT) is a new memory subsystem built for compressing main memory contents. MXT effectively doubles the physically available memory transparently to the CPU, input/output (I/O) devices, and application software. Although the memory content is in compressed form whenever possible, when the memory is accessed, the data is first uncompressed and then served to various components of the system such as the CPU and I/O devices. This prohibits the system from accessing data in the compressed form. In this paper, we present a mechanism which makes it possible to access the memory content in compressed form. Then, we show how the proposed mechanism can be used for Compressed I/O (CIO) (i.e., performing I/O operations with compressed form of data). In particular, we show that CIO can be integrated into the file system and networking subsystem. We show how MXT can be used on-the-fly and transparently for storing compressed data on the disk and transferring compressed data across the network. With respect to disk I/O, we show that our scheme can reduce the disk space requirement of the system and reduce the cost of accessing data stored on a disk. By using the compressed form of data for data transfers, we show that the observed bandwidth increases and even surpasses the maximum physical network bandwidth depending on the compressibility of data. With a typical case of 2:1 compression ratio, UDP transfer over Gigabit Ethernet demonstrates 71% improvement in network bandwidth. Our proposed scheme promises design of high performance and scalable computing systems with low cost. To the best of our knowledge, this is the first integrated solution for taking advantage of MXT technology for high performance network and I/O.

Keywords: Computer architecture, inter-processor communication, disk I/O, memory compression, scalable systems.

[1]Work presented in this paper was performed while visiting IBM T. J. Watson Research Center.
[2]To whom all correspondence should be addressed.

1 Introduction

Data compression techniques are extensively used in computer systems to save storage space or bandwidth. Both hardware and software based compression are used for storing data on magnetic media or for transmission over network links. While compression techniques are prevalent in various forms, hardware compression of main memory contents has not been used to date due to its complexity. Recent advances in parallel compression-decompression algorithms coupled with improvements in the silicon density and speed now make main memory compression practical [8, 7, 5, 11]. A Pentium based, server class system with hardware compressed main memory, called the Memory Xpansion Technology (MXT), has been designed and built [5, 11]. Results show that the performance impact of compression is insignificant on this system. Results also show that two to one compression (2:1) is practical for most applications [2, 3]. Thus, MXT effectively doubles the amount of memory in a system or alternatively provides the expected amount of memory at a smaller cost.

In the MXT system, the Compressed Memory/L3 cache controller is central to the operation of the compressed main memory [5, 11]. The MXT architecture adds a level to the conventional memory hierarchy. Real addresses are the conventional memory addresses seen on the processor external bus. Physical addresses are used between the controller chip and the compressed physical memory. The controller performs the real to physical address translation and the compression/decompression functions. The processors are off-the-shelf Pentium III processors. They run with no changes in the processor or bus architecture. Commonly used operating systems, for example Windows NT, Windows 2000 and Linux, run on the new architecture with no changes for the most part.

MXT is transparent to the CPUs, I/O devices, device drivers, and application software. Whenever, data stored in memory is accessed, it is decompressed and brought to L3 before it can be accessed by the CPU and/or I/O devices. This means that although the memory content is stored in compressed form, there is no mechanism to access data stored in memory in compressed form. Providing a mechanism for accessing data in compressed form provides a wide range of opportunities for improving the performance of I/O subsystems (for example for transferring data across the network and storing data in mass storage device in compressed form). Such a feature is essentially a natural extension of current MXT features. In this paper, we present a new mechanism called Compressed I/O (CIO) which makes it possible for applications and I/O devices to access the main memory content in compressed form. Our approach only uses the existing MXT features and requires no hardware enhancements.

In MXT systems, the memory controller translates the real addresses on the system bus to physical addresses in the physical memory. The translation is done by a lookup in the Compression Translation Table (CTT), which is also kept in physical memory. By modifying the CTT entries, it is possible to *pack* the compressed data together in real address space and make them accessible for I/O operations. This would allow us to take advantage of memory compression for network I/O and disk I/O [9, 10, 6, 4]. However, there are several obstacles which should be overcome before such an approach can be implemented and used for I/O operations.

There are several challenges involved in packing and unpacking the main memory content. One of the most significant issues that should be resolved is the mechanism through which CTT entries can be manipulated. As it will be discussed in detail in Section 3, the MXT hardware does not provide any interface through which CTT entries can be modified. Therefore, a new software approach should be used for modifying CTT entries. On the other hand, the memory controller accesses and modifies the CTT whenever data is transferred between the main memory and L3. Therefore, it is crucial to make sure that the content of CTT is in a consistent state; and software/hardware modifications to CTT (which are triggered asynchronously) do not result in a inconsistent state. Any inconsistency in the content of CTT (such as appearance of a memory sector in MXT free list more than once) can lead to fatal errors from which recovery is not possible.

Another issue which is of significant importance is providing a mechanism through which the memory controller compressors and De-compressors can be activated in a selective manner. Failure to provide such a mechanism can result in compression and decompression of already compressed or decompressed data in a

way that the original data can not be recovered.

Whenever a data buffer is packed such that it can be accessed in compressed form, we also need to maintain enough information so that the packing operation can be undone. It should be noted that when a data buffer is packed, it can be accessed in many ways. It can be copied somewhere else in the same machine. It can be sent over the network to another machine or it can can be stored on the disk. When this data is accessed later on, it has to be unpacked. The packed data itself does not have the required information for performing the unpacking. So we have to associate some additional information (i.e. meta data) with the packed data. There are several design choices for the organization and placement of the meta data and important trade offs should be studies in terms of the efficiency and size of meta data.

It is also crucial to provide rich interfaces so that packing and unpacking operations can be used by various components of the system on-the-fly. Furthermore, these interfaces should be provided in such a way that the CIO mechanism can be used transparently and there is no need for changing the application programs or operating system components. In the rest of this paper, we discuss how these challenges are addressed. The main contributions of this paper are as follows:

- We present a novel mechanism for performing I/O operations by using compressed form of data in a transparent manner by exploiting the MXT hardware.
- We discuss the basic facilities required for performing such operations along with the required infrastructure for using these facilities in both kernel mode and user level. Furthermore, we present how these facilities can be integrated into the file system and networking subsystem.
- We evaluate and analyze the performance of the basic CIO operations and the enhanced file system and networking subsystem. We show that the time for reading and writing files in such a file system is improved while the disk space requirement is reduced. We also show significant improvements in observed network bandwidth.

The rest of the paper is organized as follows. An overview of the MXT hardware is presented in Section 2. In Section 3, the basic idea behind CIO and approaches for implementing it are discussed. Mechanisms for integrating CIO into file system and networking subsystem are presented in Section 4. Various performance evaluation results are presented in Section 5. Conclusions and future research directions are presented in Section 6.

2 Overview of the MXT Hardware

The organization of the MXT hardware is shown in Figure 1. The main memory (133 MHz SDRAM) contains the compressed data. The third level (L3) cache is a shared, 32 MB, 4-way set associative write-back cache with 1 KB line size. The L3 cache is made of double data rate (DDR) SDRAM. The L3 cache contains uncompressed cached lines. It hides the latency of accessing the compressed main memory because a large percentage of accesses result in an L3 cache hit. The L3 cache appears as the main memory to the upper layers of the memory hierarchy and its operation is transparent to the rest of the hardware including the processors and I/O. The processors and I/O devices access the data in uncompressed form through the L3 cache. The L3 Cache/Compressed Memory Controller is central to the operation of the MXT system. The controller compresses 1 KB cache lines before writing them to the compressed memory and decompresses them after reading from the compressed memory.

The compression algorithm is a parallelized variation of the Lempel-Ziv algorithm known as LZ1 [8]. The compression scheme stores compressed data blocks to the memory in a variable length format. The unit of storage in compressed memory is a 256 byte sector. Depending on its compressibility, a 1 KB block of memory (which is the same size as an L3 line) may occupy 0 to 4 sectors in the compressed memory. Due to this variable length format, the controller must translate real addresses on the system bus to physical addresses in the physical memory. Real addresses are conventional addresses seen on the processor chip's external bus. Physical addresses are used for addressing 256 byte sectors in the compressed memory. The

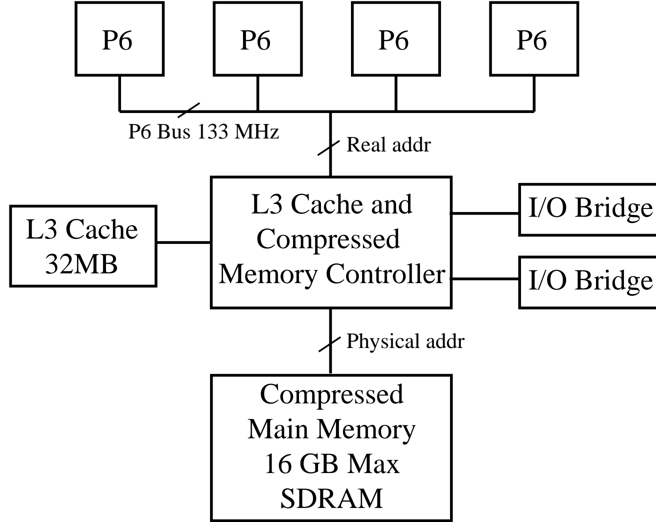


Figure 1: MXT hardware organization

real memory is merely an address space whose small sections reside in the L1/L2/L3 caches for immediate access. Rest of the memory contents reside in the physical memory in compressed form. The memory controller performs real to physical address translation by a lookup in the Compression Translation Table (CTT), which is kept at a reserved location in the physical memory. Each 1 KB block's real address maps to one entry in the CTT, and each CTT entry is 16 bytes long as shown in Fig. 2. A CTT entry includes four physical sector addresses each pointing to a 256 byte sector in the physical memory. For example, a 1 KB L3 line, which compresses by a factor of two, will occupy two sectors in the physical memory (512 bytes) and the CTT entry will contain two addresses pointing to those sectors (Fig. 2). The remaining two pointers will be invalid. For blocks that compress to less than 120 bits, for example a block full of zeros, a special CTT format called trivial line format exists. In this format, the compressed data is stored entirely in the CTT entry replacing the four address pointers (Fig. 2). Therefore, a trivial block of 1 KB occupies only 16 bytes in the physical memory resulting in a compression ratio of 64 to 1. MXT systems are configured to have real address space twice the physical memory size, since measurements show that compression ratio of two is typical for many applications [2, 3]. For example, an MXT system with 1 GB of installed SDRAM will appear as having 2 GB of memory. Maximum real address space size is 16 GB, limited by the maximum CTT size that the controller can handle.

A useful feature of the real to physical address translation is the *fast page operations*. It allows controller to swap or clear 4 KB page contents merely by updating the pointers in the CTT entries. The fast page operation clears a 4 KB page by writing the trivial line format of a zero filled block to four consecutive CTT entries corresponding to the 4 KB page. Likewise, contents of a pair of pages can be swapped by exchanging the sector pointers in their respective CTT entries. Since no bulk data movement occurs, a fast page operation runs at a much faster rate than the processor performing the same function.

As mentioned earlier, the memory content is always accessed through the L3 cache. This implies that for read operations, the memory content is first decompressed before being served to processors or I/O devices. Similarly, for write operations, the data is compressed before being stored in the physical memory. This mechanism does not allow to take advantage of compressed data for disk I/O or network I/O. In the next section, we describe a novel mechanism through which the data stored in physical memory can be directly accessed in compressed form. In the following sections, we demonstrate how our mechanism can be used for performing disk and network I/O operations in compressed form.

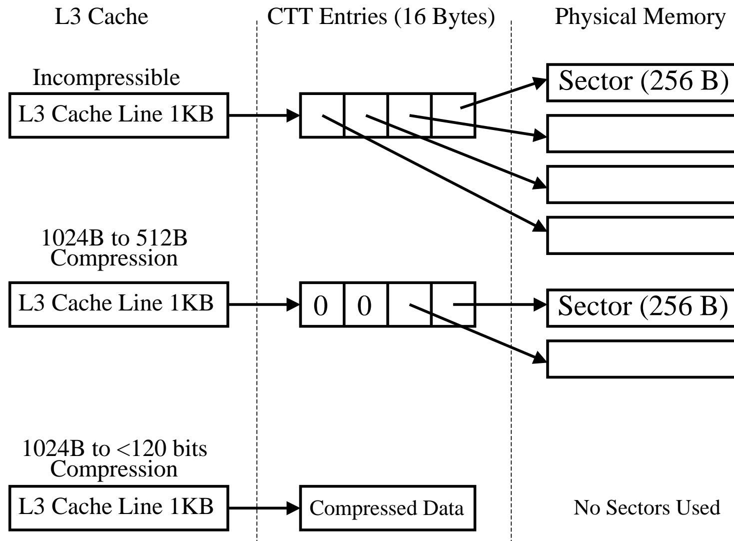


Figure 2: Real to physical address translation through Compression Translation Table (CTT)

3 A Novel Compressed I/O Mechanism

As we have mentioned before, while the IBM Memory Xpansion Technology (MXT) enables the storage of main memory content in compressed form, there is no mechanism for accessing data in compressed form. All memory accesses (including disk and network I/O operations) have to go through the shared L3 cache. For example, before the content of an I/O buffer can be transfered to network, it must be brought into the L3 cache. This means that even if the content in main memory is in compressed form, the data gets uncompressed before being sent out through the network. Similarly, when the contents of page and buffer caches are to be stored on a disk, the uncompressed form of data is used. This existing approach obviously does not allow to exploit the advantages associated with compressed data for disk and network I/O.

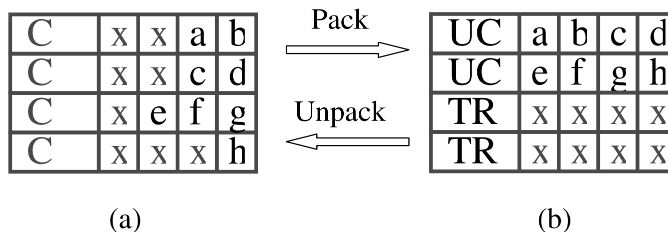
In this section, we show that we can indeed do on-the-fly compression and decompression of I/O data by exploiting the high-speed real-time compression and decompression engine in the MXT hardware. The proposed mechanism can be implemented by using only the existing features in the MXT architecture, without introducing any hardware enhancement. In this novel mechanism, which we call Compressed I/O (CIO), I/O transfers still go through the L3 cache. However, by appropriate manipulation of CTT entries, the compression and decompression units are activated selectively such that the data can be accessed in its compressed form whenever necessary.

Earlier studies have shown that in many systems the memory content can be compressed at a ratio of 2:1 or higher [2, 3]. Such compression ratios suggest that, the CIO mechanism can potentially double the bandwidth for network and disk I/O operations and can also reduce the disk space requirement by half. In the rest of this section, we first present the basic idea behind the proposed mechanism. Then, we discuss the functionalities required for supporting such a mechanism.

The idea of using both hardware or software based compression techniques for storing data on magnetic media or for transmission over network links is not new. However, the cost of hardware based techniques are usually high and the overhead of software based techniques are usually too much for general usage. Our CIO mechanism exploits the existing features of the MXT architecture. So there is no additional hardware cost. Also, as we shall see in later sections, the cost of this mechanism is very small compared with the benefit it gives to both disk I/O and network I/O.

3.1 Basic Idea

The basic idea of this mechanism is rather simple. Since the CTT is consulted for all memory accesses in the MXT architecture, it is possible to design a scheme so that data can be accessed in different forms by manipulating CTT entries.



C: Compressed Line
UC: Uncompressed Line
TR: Trivial Line
x: Null pointer
a – h: Valid pointers

Figure 3: Basic idea for CIO

The CTT entries of a memory buffer are shown in Fig. 3.a. This buffer consists of a single 4KB page or four 1KB lines. These lines are compressed into different number of sectors. Overall, eight 256B sectors are used for storing the content of this buffer in compressed form. Consider modifying the CTT entries so that the valid pointers are packed together in the beginning of the CTT entries as shown in Fig. 3.b. We also change the control bits in these CTT entries to indicate that these lines are in uncompressed form (marked as UC in the figure). We then set the CTT entries of other lines such that they represent trivial lines (marked as TR in the figure). After these modifications, we can essentially read the data in compressed form by accessing the first 2 KBytes of this memory page. (We will discuss the cases where the number of used sectors is not a multiple of four in Section 3.3.) We call the set of operation required for accessing the content of a memory buffer in compressed form the *Pack* operation.

Once a buffer is packed, we can modify the CTT entries so that the original layout of CTT pointers and other state information are restored. This operation is called *Unpack*. Suppose a sender process wants to send this buffer over the network. It can first pack the buffer, and transfer it to the receiver. After the receiver gets the packed data, it will unpack it and store it in the memory. Whenever this buffer is accessed, the data will be brought into the L3 cache, after being automatically uncompressed by MXT hardware.

There are several challenges involved in implementing the Pack and Unpack operations. First, we need to have a way to read and modify the CTT entries. As mentioned in Section 2, the MXT architecture has several CTT manipulation operations implemented in hardware as fast page operations, including one to read the CTT entries of a memory page in real address space. However, there is no hardware support for modifying CTT entries directly. Second, when I/O data is packed, we want it to appear as uncompressable data to the hardware such that it does not get compressed again. Otherwise the packed data may get compressed again and lead to a change in the order of used sectors when data is decompressed. This will lead to an incorrect unpacking of data. Therefore, it is crucial to deal with this requirement properly. Third, we need to maintain enough information so that the Unpack operation can be performed successfully. After the data is packed, it can be accessed in many ways. It can be copied somewhere else in the same machine. It can be sent over the network or it can can be stored in the disk. When we access the data later, they have to be unpacked on the same machine. The packed data itself does not have the required information for performing the Unpack operation. So we have to associate some additional meta data with the packed data. There are several design choices for the organization and placement of the meta data. In the following subsections, we will discuss how these issues can be addressed.

3.2 Reading and Modifying CTT Entries

The CTT in the MXT architecture introduces a level of indirection for memory accesses. This provides much flexibility for manipulating the content of memory. Fast page operations in the MXT architecture, such as swapping pages and zeroing pages, take advantage of this flexibility. These fast page operations, perform operations such as move in a very efficient manner by modifying CTT entries in hardware. There is also a fast page operation to read the CTT entries for a given page. However, we need a more general way to update CTT entries than that provided by fast page operations. Since there is no direct support in hardware for such operations, CTT entries can be manipulated by exploiting the software CTT *loopback* mechanism in the MXT architecture.

The CTT loopback mechanism essentially enables us to access the CTT entries just like normal memory. In MXT machines, the CTT table is placed in physical memory, just like normal code and data. In the real address space, for each line, the pointers in the corresponding CTT entry point to the sectors allocated for that line. There are CTT entries for the whole real address space. The CTT loopback mechanism sets up a region of real address space such that its corresponding CTT entries point to the CTT itself. This mechanism is illustrated in Fig. 4.

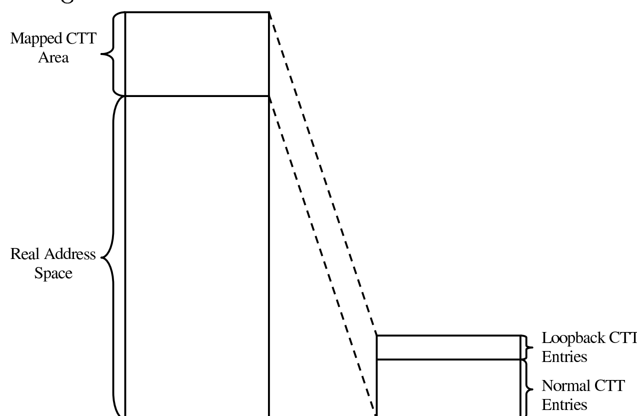


Figure 4: CTT loopback mechanism

The CTT loopback cannot be accessed directly by the operating system because it is above the maximum real memory. The CTT loopback can be re-mapped such that it can be accessed for both read and writes. However, great caution must be taken in using this feature, especially when CTT entries are modified by using the CTT loopback. Any invalid entry in CTT table may result in crashing the system.

It should be noted that when CTT entries are accessed through the CTT loopback, they will be brought into the L3 cache. Since the hardware may also update CTT entries, we must make sure that the CTT is always in a consistent state. When a CTT entry is being modified through the CTT loopback, the hardware should not modify that CTT entry or any other CTT entry which falls in the same L3 cache line. To satisfy this requirement, we can use a set of special buffers which we call CIO (Compressed I/O) buffers. CIO buffers should be 64K bytes aligned, and their sizes are multiples of 64K bytes. CIO buffers are reserved only for pack and unpack operations and are not accessed by any other part of the system. An L3 cache line (1KB) of CTT entries represents 64K bytes address range in the real address space. (Recall that each CTT entry which is 16 bytes long represent 1K bytes of real address space.) Because of the size and alignment of CIO buffers, no CTT entries in the same cache line will be accessed by the system hardware. As we will discuss in the next section, for packing and unpacking the content of a buffer and to avoid any inconsistence in the CTT, the data can be first swapped or moved to a CIO buffer by using fast page operations.

After modifying CTT entries, these CTT entries must be flushed out of L3 before they can be effective. This can be achieved by using a fast page operation. We have to also ensure that the CTT entries are always in uncompressed form. Another feature of the MXT architecture called Compression Disable (CD) region can be used to achieve this goal. When an L3 cache line is evicted, if the real address space it represents is in the CD region, the MXT hardware will not attempt to compress it before storing it in main memory. At any given time, we can mark one region of the real address space as the CD region. When a CTT entry

is being modified, the portion of CTT loopback which contains this entry should be put in the CD region. When the L3 cache line containing this CTT entry is flushed or evicted from the L3 cache, it will be stored in physical memory uncompressed.

3.3 Accessing Data in Uncompressed Form

After the data is packed, whenever it is accessed it will be brought into the L3 cache. The packed data should appear as uncompressed data to the system such that it does not get compressed when the corresponding L3 lines are evicted from the cache. This goal can be achieved by modifying the CTT entries to indicate that the lines which contain the packed data are uncompressed. Since each uncompressed line must have four valid pointers, a boundary condition exists where the number of sectors for the compressed data is not a multiple of four. To solve this problem, a set of *padding sectors* can be used to pad the last line if needed. The MXT hardware maintains a list of free sectors in the physical memory. By modifying CTT entries directly for a special CIO buffer, we can *steal* sectors from the system. These sectors are not used in the system and will be removed from the free list. The pool of padding sectors can have a high watermark and a low watermark. When the high watermark is reached (indicating that there are too many padding sectors) some of the padding sectors can be returned to the system. This also can be done by modifying CTT entries for a special CIO buffer. When the low watermark is reached, more free sectors are stolen from the MXT free list.

When the data is about to be unpacked, we have to make sure that the data is flushed from the L3 cache to the main physical memory. This can be done by using flush and invalidate fast page operation. Since the data is packed and is in compressed form, they should not be compressed again when the flush happens. Otherwise the sector layout will not be similar to that when they were packed. To achieve this, we can use the CD region feature of MXT. Before the flush, the data of interest should be put under the CD region. This way the data will be stored in uncompressed form in physical memory. In some cases, the data may have already been flushed or evicted from L3 before the CD region is set up appropriately. In these cases, after setting up the CD region, the data should be touched so that it is brought into the L3 cache. Then it should be marked as dirty. Now, when the data is flushed out of L3 again, it won't be compressed. The flushing operation is performed on a page by page basis. For the memory pages in the CD region, each line will have four sectors after the flush. For the last page, we may end up with some sectors which don't contain the packed data. These sectors can be added into our padding sector pool during the Unpack operation.

3.4 Meta data

In order to unpack the data successfully, we need to reconstruct the CTT entries for the data. The packed data itself doesn't contain the required information for doing so. Therefore, the packed data should be associated with additional information (i.e., meta data) required for successfully unpacking it. The required meta data can be constructed during the pack operation. The organization of the meta data should facilitate the process of packing while it does not increase the cost of packing significantly.

The natural choice for the organization of meta data is to have each 1KB of data to be represented with the required information which contains for example the number of sectors associated with the line. The information for each 1KB line can be of constant or variable size. While using variable sizes for different elements of the meta data can reduce the overall size of the meta data, representing each line of data with elements with constant size can make the Unpack operation more efficient.

Since the objective of meta data is to construct the CTT entries during the Unpack operation, the CTT entries of the ordinal data can be used as a start point for meta data. For trivial lines, copies of the original CTT entries can be used. For non-trivial lines, exact copies of the original CTT entries cannot be used because the real values of the sector pointers in the original entries may not be meaningful at the unpack time. For example, the unpack operation can be performed on a system other than the one in which the Pack operation was performed. Therefore, these pointers can be changed to indices which represent the

position of these sectors in the packed data. During Unpack, this indices can be changed back to valid sector pointers which point to the physical location of the corresponding sectors.

The meta data can be stored either with the data or separately. When an I/O subsystem which is CIO aware uses packed data, its meta data can be used as protocol data and placed separately with the I/O data. For example, during the pack operation the meta data can be stored preceding the packed data. Thus, I/O subsystems can blindly transfer the meta data along with the data without distinguishing them or without even being aware of the fact that the data is in packed form.

4 Implementation

In this section, we first discuss the design and implementation of several basic operations for the packing and unpacking process. They serve as building blocks for more specialized interfaces. These basic operations are implemented as Linux kernel functions in a module. Thus they can be used by network or I/O protocols in the kernel. However, we also provide a method to export this functionality to the user space. In this case, these operations can be incorporated into user level I/O or communication protocols, or they can be used as wrapper functions to enable transparent on-the-fly compression and decompression in some legacy protocols such as UDP. To demonstrate the benefit of this mechanism, we have implemented it both for disk I/O and network I/O. For disk I/O, we implemented a new compression and decompression module for the e2compress file system in Linux [1]. It uses the kernel interface of the CIO mechanism. For network I/O, we exploited the user level interface by using the CIO mechanism to do on-the-fly compression and decompression for UDP datagrams.

4.1 Meta data Format

The meta data format we have chosen is a relatively simple one. There is one meta data entry for each line of the data buffer. For trivial lines, the meta data entries are exact copies of their CTT entries. For non-trivial lines, the meta data entries are also copies of their CTT entries with the exception that the CTT pointers are changed to indices which represent the positions of these sectors in the packed data.

When we put meta data and packed data together, the receiver needs to know the length of the meta data in order to do unpacking successfully. This is done by taking advantage of several unused bits in the CTT entries. Basically the number of pages packed are encoded into the first two entries of the meta data, and the length of the meta data can be derived from this.

Another problem arises when we want to put meta data and packed data together. In order to do unpacking successfully, we need to make sure the beginning of the packed data is sector size aligned. The meta data comes before the packed data. So we always round up the size of meta data to multiples of sector size, which is 256 bytes in current MXT machines. In this way if the receiving buffer is sector aligned, the packed data will also be sector aligned.

4.2 Basic Operations

There are three basic operations for our CIO mechanism. They are Pack, Unpack, and Unpack_local. The Unpack_local is essentially a fast version of Unpack. It can only be used when the packed data is in the original buffer instead of being copied or received from somewhere else.

We use network I/O as an example to illustrate the significance of these basic operations. Next, we describe the steps involved in these basic operations. In the following subsections, we show how these basic operations can be used for implementing disk I/O and network I/O interfaces.

A typical scenario of network I/O using the proposed CIO mechanism will be as follows:

1. At the sender side, the sender calls Pack to compress the I/O buffer. The packed data is moved or swapped to a CIO buffer first, and moved or swapped out to the output buffer when the packing is

done. They can also be sent directly from the CIO buffer.

2. The packed data is sent to the receiver.
3. The sender calls Unpack_local to restore the original content of the I/O buffer.
4. At the receiver side, the receiver grabs a CIO buffer and prepare it for receiving data.
5. Data is received into the CIO buffer.
6. The receiver calls Unpack to uncompress the data and put them into the receive buffer.

The Pack function does the major work of packing the I/O buffer and putting it into a CIO buffer. Currently, the I/O buffer has to be a set of physical pages. The Pack function consists of the following steps:

1. Swap or move the source pages with the pages in CIO buffer. This is done by using the swap or move fast page operation. This operation also flushes and invalidates the pages from the cache.
2. Put Compression Disable (CD) area over the CTT entries for the CIO buffer.
3. Make a copy of CTT entries of the CIO buffer.
4. Do the packing. This step generates the meta data and updates the CTT entries of the CIO buffer.
5. Flush and invalidate the CTT entries from the cache.
6. Copy the meta data before the CIO buffer.

The Unpack function does the major work of unpacking the data in the CIO buffer and putting them into the destination buffer. Currently, the destination buffer also has to be a set of physical pages. Unpack consists of the following steps:

1. Make a copy of the meta data.
2. Flush and invalidate the CIO buffer from the cache.
3. Unpack the data. This step uses the meta data to generate the correct CTT entries for the unpacked data in the CIO buffer.
4. Put CD area over the CTT entries of the CIO buffer.
5. Update the CTT entries. Flush and invalidate the CTT entries from the cache.
6. Swap or move the pages to the destination buffer.

After we have called Pack, the content in the source pages are no longer valid due to the swap or move operation. Sometimes we need to restore the ordinal content of the source pages. It can be done by calling the Unpack function at the sender side. However, this is not necessary because the pointers in the CTT entries before we do the packing are the same as those after we restore the content. So here we use a copy of the original CTT entries to do this quickly. This is done in function Unpack_local. This function uses a saved copy of the original CTT entries to save the time of flushing and going through the meta data analysis.

We should also note that before the the message is received into the CIO buffer, some preparation work must be done. Otherwise the data received may be evicted or flushed out of the L3 cache and got compressed a second time before we unpack them. To fix this we can mark the CIO buffer as CD area before the data is received.

The three functions presented here serve as basic building blocks or starting point for more specialized interfaces. They can be viewed as a kernel service provided to the kernel programmer. However, as we shall see, they can also be exported to the user space. The actual interface for a certain I/O protocol may be slightly different due to the alignment and meta data placement requirements. In the next subsection, we discuss how this basic interface can be tailored to disk I/O and network I/O interfaces, respectively.

4.3 Disk I/O Interface

To study the impact of our CIO mechanism on disk I/O interface, we implemented a new compression and decompression algorithm for the e2compress file system under Linux. In this implementation, we essentially used the kernel interface of our basic operations with only minor changes to make them satisfy the interface requirements for the e2compress file system.

4.3.1 E2compress File System

The e2compress file system is an extension to the commonly used Linux ext2 file system [1]. It performs compression when the data is stored to the disk, and decompression when the data is fetched from the disk.

In this file system, only regular file data can be compressed. File system meta data, such as super blocks, i-nodes and directories, are never compressed. Access to compressed data is done transparently by the file system. The compression algorithm can be specified on a per-file basis.

In the e2compress file system, the basic units of compression and decompression are clusters. The size of a cluster is a multiple of the disk block size. During the compression, the file is compressed cluster by cluster. The actual sizes of compressed clusters must also be multiples of the size of a disk block. We can see that internal fragmentation may happen here if the compressed size is not a multiple of disk block size. And if we increase the cluster size or decrease the disk block size, the compression ratio should improve. The cluster size can also be specified on a per-file basis.

In the e2compress file system, we can also specify the compression algorithm and cluster size for a directory. In this case, new files created in this directory will be automatically compressed with the algorithm and cluster size specified for the directory.

4.3.2 Implementing the MXT Algorithm for E2compress File System

The e2compress file system can be extended to use new compression and decompression algorithms. Currently, only software algorithms such as gzip and lzv are used. We implemented a new MXT algorithm which uses our Pack and Unpack functions as a Linux kernel module and integrated it into the e2compress file system. Since the compression and decompression must be done in the Linux kernel as part of the file system, we used the kernel interface of our CIO mechanism. Basically, the Pack operations serve as the compression algorithm and the Unpack operations serve as the decompression algorithm.

In the file system, the I/O buffers are always page aligned. Thus the alignment requirement of our functions is not a problem. However, the interface is still slightly different from our basic operations. First, the e2compress file system copies the data buffer to a working area before doing compression and decompression, and the results are stored to another buffer. So, it is not necessary to restore the ordinal buffer after compression and decompression. In other words, there is no need to call the Unpack_local function. Second, a small space must be reserved at the beginning of the buffer because it is used by e2compress file system itself to store such information as compression algorithm used, compressed length, etc. Third, the results must be put in buffers specified by the system instead of our CIO buffer. This can be done by using swap or move fast page operation. Thus for e2compress file system we need two functions: mxt_compress and mxt_decompress. They correspond to Pack and Unpack functions. The difference is that they also satisfy the above constraints.

Our MXT algorithm is implemented in two Linux modules. The first module provides the two functions above. The second module is part of the e2compress file system and it calls the these functions to do the compression and uncompression work.

4.4 Network I/O Interface

To demonstrate the potential of increasing the effective network data transfer bandwidth by using our CIO mechanism, we also implemented an interface for the compression and decompression of UDP datagrams.

In this work, we used the user space interface of our basic operations.

4.4.1 Using CIO to Wrap UDP Socket Functions

In order to do the compression and decompression of UDP datagrams transparently, we took an approach of wrapping the datagram sending and receiving socket functions with our CIO operations. The approach was implemented as follows.

The socket function to send a datagram is *sendto*. Before calling this function, we check the message address and length, and call the Pack function to compress the data. Then we send the message with the updated length. After that, we call the Unpack_local function to restore the content of the original message. At the receiver side, we also change the socket function *recvfrom*. After this function, we call the function Unpack to decompress the data and return the actual length of the unpacked message.

The above changes to the socket function can actually be done automatically. Also the changes are not limited to UDP socket functions. It is possible to use this wrapping approach to enable compression transparently for other networking protocols. For example, it can be used in MPI [9, 10] to enable on-the-fly data compression and decompression over the network. However, our pack and unpack functions require that both the send and receive buffer be page aligned. If the buffers can be anywhere in the user space, an extra copy of data may be necessary.

4.4.2 Exporting the Interface to User Space

The interface to user space network I/O is different because compression and decompression must be done in user space and the buffer address is virtual address, while basic Pack and Unpack functions are implemented in kernel and work on physical pages. To export them to the user space, we implemented a dummy character device in Linux. By calling ioctl function on this dummy device, the user program can trap into the kernel, do the address translation from virtual addresses to real addresses and call our pack and unpack functions.

Another problem is that contiguous buffers in user space may not be contiguous in real address space. At the sender side, we swap them page by page to a CIO buffer before we do the packing. At the receiver side, we also want to swap the buffers page by page to a CIO buffer. However, since the swap fast page operation also flushes the pages, we need to mark the the buffer as a CD area in order to avoid accident hardware compression. Since the CD area only works for contiguous physical pages in real address space, we need to do it also on a page by page basis.

We implemented the CIO mechanism for network I/O in two parts. The major part is a kernel module which implements the Pack and Unpack operations and provides a dummy character device to access them. The address translation from user virtual addresses to physical addresses is also done in this modules. The other part is a user library which encapsulates the ioctl functions to the dummy character device. This library can be used by user programs to access the CIO functionality.

5 Performance Evaluation

In this section, we present the performance results for our CIO mechanism. We used two 1GHz MXT-capable machines for performing our experiments. These machines were running Linux 2.2.14 kernel and were equipped with 512 MB of physical memory and Acenic Gigabit Ethernet network adapters. The hard disk used in these machines were IBM Ultrastar SCSI disks. In the rest of this section, we first present the performance for the Pack and Unpack operations. Then, we discuss the performance results obtained from using CIO for implementing disk I/O and network I/O operations, respectively.

5.1 Pack and Unpack Operations

For measuring the cost of the basic Pack and Unpack operations and their breakdowns, we instrumented our code by using the time stamp counters in the MXT machines. The timing breakdowns for packing and unpacking one page are shown in Fig. 5. These results are obtained for three different compression ratios as shown in this figure.

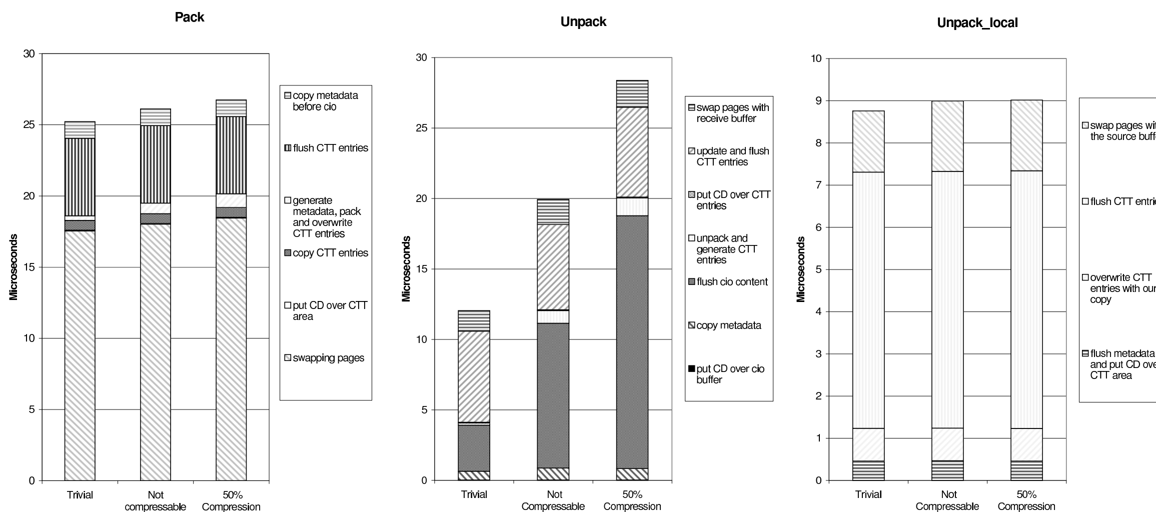


Figure 5: Timing breakdown of basic operations for one page

First, let us take a look at the timing breakdown for the Pack function. It can be seen that the major component of the cost is the cost of swapping pages to the CIO buffer and flushing and invalidating CTT entries. The swap operation is actually performed in two steps. In the first step, the memory pages are invalidated and flushed from the L3 cache. In the the second step, the CTT entries of the two pages involved in this operation are swapped. The cost of invalidating and flushing memory pages is much more significant than the cost of swapping CTT entries. The speed of flushing a page from L3 is limited by the hardware. When a memory page is flushed, the hardware compression engines are triggered, and corresponding four L3 cache lines are compressed and stored in physical memory.

It can be observed that the second major component of the Pack operation is related to updating CTT entries. For making modifications made to CTT entries through the CTT loopback, the modified entries should be invalidated and flushed from L3. Hardware support for performing such an improvement (for example by providing a fast page operation for writing CTT entries) will significantly reduce the cost of modifying CTT entries. Such a hardware support will also make it unnecessary to use a special CIO buffer for manipulating CTT entries.

Similarly, the major portion of the cost of Unpack is the cost of flushing and invalidating the received message in the CIO buffer and also modified CTT entries. It can be seen that the cost of flushing the CIO buffer depends on the compression ratio.

It can be observed that the cost of Unpack_local is much less than that of Unpack as a saved copy of CTT entries are used to restore the original content of the CTT entries. It can be also observed that the major component of cost of Unpack_Local is the cost of flushing and invalidating the CTT entries from L3 cache.

It should be noted that the cost of flushing CTT entries doesn't increase linearly with the number of pages being packed. Since the size of CTT entries for a page is only 64 bytes, they tend to stay on a single cache line even when we increase the number of pages. As it can be seen in Fig. 6, the cost of packing does not increase linearly with the number of pages.

From the discussion above, we can see that the Pack function is the most time-consuming operation. However it can be observed that, the maximum bandwidth of the packing operation is more than 200

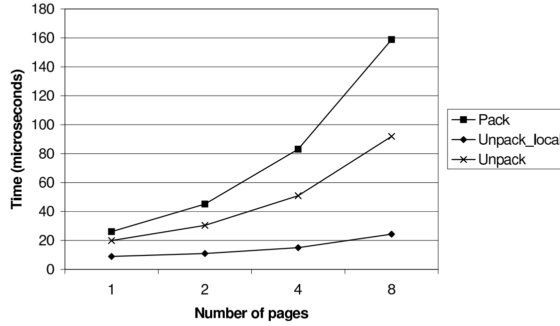


Figure 6: Cost of basic operations for different number of pages

MB/s for large data sizes. This number is still higher than the bandwidth of most current disk systems and networks. So it can be expected that when the CIO mechanism is used for I/O operations, the cost of packing and unpacking will not become a limiting factor in achieving the maximum disk and network bandwidth possible. In the following subsection we evaluate the performance of CIO operations for disk and network I/O.

5.2 Disk I/O

To evaluate the usage of CIO with Linux ext2 file system (enhanced with e2compress), we measured the cost of accessing the files stored in such a system. For our evaluation we used both synthetic and real files of various sizes. We used synthetic files such that we could control the size of compressed files when CIO or software compression methods are used.

We created a number of synthetic files with different sizes. These synthetic files are created in such a way that half of the file content cannot be compressed and the other half can be compressed to trivial lines. Thus in memory, these file can be compressed with a ratio of about 2:1. However, because files have to be stored in blocks on disk and e2compress file system also stores extra information on disk for a file, the actual compression ratio on disk may be lower. The advantage of using such files is that the size of compressed files were essentially the same regardless of the type of the compression method used (for example MXT hardware compression and gzip software compression methods). This way, the size of compressed file did not vary while the cost of accessing these files were measured.

The default compression and decompression algorithm in the e2compress file system is gzip. This refers to the algorithm used for compression and decompression, not the commonly used GNU zip utility. The gzip algorithm is software based and its implementation is considered to be quite efficient. So we chose it as a basis for the comparison.

Fig. 7 shows the cost of compression and decompression of files with hardware MXT and software gzip methods. The notation BmCn in the figure represents the case where the disk block size is mKB and the cluster size is nKB. Two file sizes (32MB and 128MB) and different cluster sizes and disk block sizes were used to obtain these results. These experiments were carried out in such a way that reading and writing files happened in the system file cache and there were no disk activities. For these synthetic files, the compressed sizes were roughly the same for both the gzip and the MXT methods. It can be seen that the MXT algorithm is much better than the gzip algorithm in terms of compression time. The factor of improvement is from 301% to 340%. For decompression, the MXT algorithm does not have such a big advantage, but it is still 10% to 28% faster.

For dictionary based software algorithms like gzip, usually decompression is much faster than compression. And the content of the data has to be checked during both compression and decompression. Our MXT algorithm, on the other hand, is more balanced since the compression and decompression time are about the same. Also, the CIO mechanism only checks and updates the CTT entries. So the compression and decompression are achieved without touching the data.

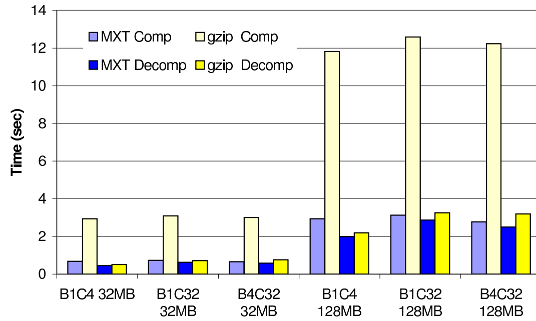


Figure 7: Cost of disk I/O without disk access

We should note that the performance numbers in Fig. 7 include not only the compression and decompression time, but also other overheads in the ext2 (and e2compress file system). We also instrumented our code to get the time spent in MXT compression and decompression routines (which perform the Pack and Unpack operations). For example, the time spent for compressing the 128MB file in the B4C32 case with MXT method was measured to be roughly 8 seconds. This translates to 25 microseconds for each page which is quite close to the numbers presented in Fig. 5. It should be noted that the cost of accessing smaller files were also measured. Those results show a similar trend and are not presented here.

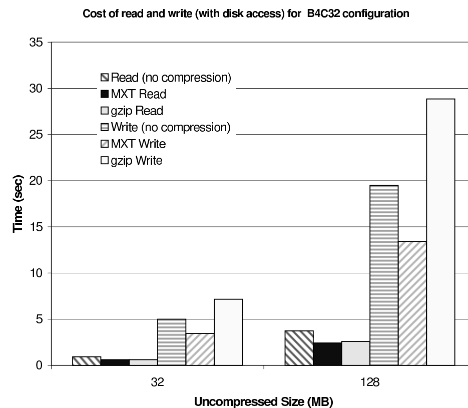


Figure 8: Cost of disk I/O with disk access for 32KB cluster size and 4KB block size

The access times for 32MB and 128MB were also measured when the files were read from or written to a disk. For these experiments we also compared the performance of compressed file system with that of non-compressed file system. The results for the case where block size was 4 KB and cluster size for compressed file system was set to 32 KB are shown in Fig. 8. Both reading and writing of files were performed synchronously. For reading a file, we ensured that the content was not in the file cache before the test. For writing a file, we made sure that the file content was indeed written to the disk (by invoking the *sync* command).

It can be seen that for reading, due to the reduced file size, both MXT and gzip perform better than the case where compression is not used. MXT is up to 6.7% faster than gzip. For writing, the MXT algorithm is the best of the three approaches because it does the required compression operations very efficiently, and files are stored in fewer disk blocks compared with the no compression case. It outperforms gzip by from 108% to 115%. The gzip algorithm performance is the worst for write operations. Although gzip benefits from using fewer disk blocks for storing a file, it spends a significant amount of time doing the compression.

We should note that in these tests, the synthetic files we used have a very simple pattern. It is possible that compression and decompression can be done slower in more complex patterns for software compression algorithms. Also, the e2compress file system will store files with the original content if it finds that it cannot be compressed. So during file reading, half of the file content comes directly from the disk and does not go through the decompression routine at all. Such conditions may not arise for real files.

Table 1: Results for real files with block size 4KB and cluster size 32KB

	linux.tar			db2_real1.del			db2_real2.del		
	MXT	gzip	Non	MXT	gzip	Non	MXT	gzip	Non
Size (KB)	57924	33488	89816	44104	15052	119520	1928	652	5116
Read (s)	3.119	3.938	3.163	3.872	3.913	4.206	0.169	0.173	0.203
Write	10.347	32.812	13.306	8.723	16.577	17.694	0.413	1.293	0.779

We also used some non-synthetic files under e2compress enhanced file system to evaluate the performance of CIO for disk I/O operations. The results are shown in Table 5.2. The file linux.tar is a tar file of the linux source directory on one of our MXT machines. It also includes the object files generated after compiling. db2_real1.del and db2_real2.del are two database files taken from an IBM DB2 database. The time measured includes disk access time. The writing time is broken into two parts. The first one is compression time or copying time for the case with no compression. The second time is the time spent in *syncing* the file cache. From the table it can be seen that our MXT mechanism outperforms other methods very well. For the read operation, it is up to 26% faster than the gzip algorithm. For the write operation, it is up to 213% faster than gzip. The gzip algorithm now spends a large amount of time doing the compression (up to 11 times more than MXT) due to more complex file patterns. It can also be noticed that gzip has a better compression ratio for these files. This is because the MXT hardware compression works on 1k lines. When we use a quite large cluster size (32 K bytes), the MXT algorithm doesn't benefit much in term of compression ratio.

5.3 Network I/O

In our network I/O test, we measured the stream bandwidth of UDP datagrams between two MXT machines connected with Gigabit Ethernet. The client machine keeps sending UDP datagrams to the server. The server machines enters a loop and stops after it has received a certain number of datagrams. The bandwidth numbers we got are shown in Fig. 9.

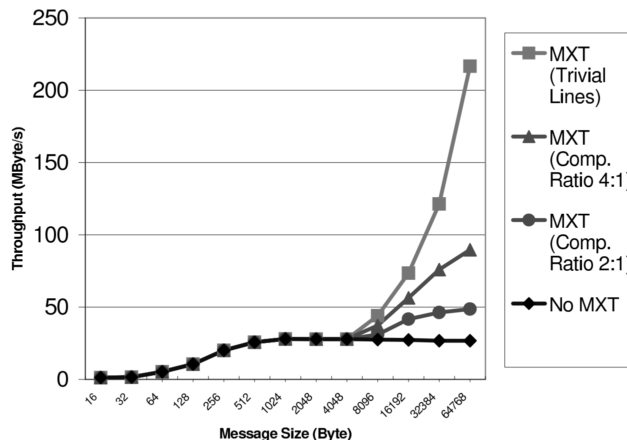


Figure 9: Bandwidth for UDP data transfer with different compression ratio

In these experiments, both the source buffer and the destination buffer were page-aligned. For message sizes up to 4096 bytes, MXT compression was not used and data was transferred in uncompressed form. It can be seen that with no compression, the UDP bandwidth quickly saturated at about 28 MB/s. With the use of MXT compression and decompression, for the typical case of 2:1 compression ratio, the peak bandwidth is increased to more than 48 MB/s. Thus the factor of improvement is over 71% in this case. If the buffers can be compressed to all trivial lines, then the effective peak bandwidth is more than 200 MB/s. That's even more than the bandwidth of the physical link. If the compression ratio is 4:1, the effective peak bandwidth is about 90 MB/s.

6 Conclusions and Future Work

In this paper, we have presented a novel mechanism called Compressed I/O (CIO) for performing I/O operations in compressed form by using the MXT architecture. We have shown how two basic operations called Pack and Unpack can be used to access data stored in main memory in compressed form. Furthermore, it has been shown how these operations can be used to improve the performance of the Linux ext2 file system (enhanced with e2compress addition for supporting compressed files) and the networking subsystem (i.e., the socket interface for UDP). The performance of the file system and the networking subsystem is shown to improve significantly by using CIO.

We are currently investigating how the performance of the CIO mechanism can be improved in particular with providing additional hardware support (such as new fast page operations for modifying CTT entries). We are also investigating how the CIO mechanism can be used in building scalable communication and I/O subsystems for clusters. We also plan to evaluate the impact of CIO on the performance of various applications.

Disclaimer

The CIO implementation presented in this paper is not a part of any IBM product and no assumptions should be made regarding its availability as a product in the future.

References

[1] E2compress (e2compr) File System. http://www.netspace.net.au/ reiter/e2compr/.

[2] B. Abali, M. Banikazemi, X. Shen, H Franke, D. E. Poff, and T. B. Smith. Operating System Support and Performance Evaluation of Hardware Compressed Main Memory. *IEEE Transactions on Computers.* In press.

[3] B. Abali, H. Franke, D. E. Poff, X. Shen, and Smith T. B. Performance of Hardware Compressed Main Memory. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'01)*, pages 73–81, Jan 2001.

[4] A. Acharya, M. Uysal, and J. Saltz. Active Disks: Programming Models, Algorithms and Evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* , pages 81–91, October 1998.

[5] S. Arramreddy, D. Har, K. Mak, T. B. Smith, B. Tremaine, and M. Wazlowski. IBM X-Press Memory Compression Technology Debuts in a ServerWorks NorthBridge. In *Proceedings of HOT Chips 12 Symposium*, Aug 2000.

[6] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach.* The IEEE Computer Society Press, 1997.

[7] P. Franaszek, P. Heidelberger, and M. Wazlowski. Management of Free Space in Compressed Memory Systems. In *Proceedings of the ACM Sigmetrics*, 1999.

[8] P. Franaszek, J. Robinson, and J. Thomas. Parallel Compression with cooperative dictionary construction. In *Proceedings of the IEEE Data Compression Conference (DCC'96)*, pages 200–209, June 1996.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.

[10] Marc Snir, Steve Otto, Steve Huss-Lederman, David Walker, and Jack Dongarra. *MPI–The Complete Reference. Volume 1 - The MPI-1 Core, 2nd edition.* The MIT Press, 1998.

[11] 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(2):56–68, March/April 2001.