RC 21808 (98136) 19 July 2000 **ComputerScience**

IBM Research Report

Improving the Performance of the Tivoli Storage Manager with Threaded Hardware Compression

Jian Yin
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120

Mazin Yousif
IBM Research Triangle Park
3039 Cornwallis Road
RTP, NC 27709

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

Improving the Performance of the Tivoli Storage Manager with Threaded Hardware Compression [1]

Jian Yin +
Mazin Yousif Σ
Ronald Mraz *

+ IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Σ IBM Research Triangle Park
3039 Cornwallis Road, RTP, NC 27709

* IBM T.J. Watson Research Center
Route 134 and Kitchawan Roads, Yorktown Heights, NY 10598
mraz@us.ibm.com

Abstract

The ADSTAR Distributed Storage Manager (now the Tivoli Storage Manager) is a client/server system that provides backup and storage archival functionality to customers in multi-vendor computer environments. To improve the performance of the ADSM system this paper describes experimental results for an ADSM production product enhanced with a hardware based data compression system built of commodity components. We demonstrate that threaded hardware compression improves the performance of ADSM by 2X while reducing the CPU loading by at least one-half, when using high bandwidth network communications. Previously, the CPU load was at 100% forcing the user's client offline for backup purposes. The method described can also be applied to multi-threading software data compression across multiple CPUs in a shared or distributed memory parallel program environment. Furthermore, this methodology can be applied to many other compute intensive algorithms, such as data encryption, that limit the performance of client/server systems.

Subject Areas: Innovative hardware/software tradeoffs, High-Performance I/O architectures, Interconnection networks and network interfaces, Novel architectures for emerging applications, Simulation and performance evaluation.
Keywords: Multi-threaded hardware accelerators, Backup, Restore, archival systems, e-Commerce, web serving.

[1]Student Paper Submission - Jain Yin did work as summer hire at IBM Almaden Research Center. Submission to the Seventh International Symposium on High-Performance Computer Architecture, January, 2001. Rev. 1

1 Introduction

The goal of this study is to improve the performance of the IBM ADSM [2] product [5] by offloading the backup client's data compression function to a hardware accelerator card. We observed that hardware compression improves ADSM backup performance by three times over the existing software compression implementation. This improvement includes the time that ADSM must buffer and DMA this data through the PCI interface for access to the compression hardware. Moreover, hardware compression reduced the CPU loading by at least half in all experimental studies.

The main motivation for supporting data compression is that reducing the size of backup data, through compression, helps the overall backup system performance. Compression reduces the required space to store the backup data in ADSM servers which allows the backup server to handle compressed data more efficiently than when uncompressed. Additionally, compression reduces the data transferred on the network between ADSM clients to ADSM servers, and hence has the effect of making any network appear faster. However, there are trade-offs since the compute intensive nature of data compression can be a performance bottleneck and potentially degrade the performance of other services and applications running on the same system. This can limit system scalability if the backup server performs compression (or any other compute intensive operation) and attempts to service multiple clients simultaneously.

To insure scalability of the ADSM client/server system, (i.e. the ability of a single server to service 100's to 1000's of simultaneous clients) all compute intensive data operations are performed within the client. One of the most compute intensive operations that a client can perform is data compression of the archival data. Software compression limits performance of the client/server system when fast networks, such as 1000 Mbit/sec Ethernet, are used for data transfers. This is because, at a compression rate of 1-2 Mbytes per second in a 133Mhz machine, it takes longer to compress the data than to send the data across networks capable of sustaining transfer rates 10X the speed of software data compression.

This paper provides a system overview and reports experimental results produced by augmenting a product level version of ADSM with the ability to use a hardware compression accelerator. Section 2 provides a description of the software compression system as it applies to ADSM. Section 3 describes our hardware compression implementation and it's integration into ADSM. Section 4 describes or experimental results for backup and archival using no compression, software compression and hardware compression. Section provides comparison using actual client data with high end client server hardware chosen to eliminate any potential bottlenecks in the datapath. Section 6 provides our Conclusions and Recommendations for future work.

2 Software Data Compression in ADSM

The ADSM backup client has five stages of operation and uses these stages to exploit parallelism in computer system components of storage, CPU and network. These pipeline stages are: user authentication, read data, optional data compression, send data, and the final commit of data to archival storage. To exploit the backup system's pipelined parallelism, the compression is performed in "chunks" of data, one buffer at a time. In this way, while the CPU is performing compression, local storage can be read for the next chunk of data and network transfers can be overlapped.

To archive data, the ADSM backup client reads the datafile in chunks (Typically set at 32K bytes per chunk) and passes each one to the compression routine. The compression routine compresses the data in the input buffer and places the result in the output buffer. Multiple chunks of input data are compressed, in this way, until the designated output buffer is full. When the output buffer is full, ADSM begins the send

[2]ADSM is now the TSM (Tivoli Storage Manager) ADSM is a shorthand for ADSTAR Distributed Storage Manager, which is a client/server program that provides storage management solutions to customers in multi-vendor computer environments.

data stage, which transfers the data in the output buffer to the remote backup server. The program then repeats this process until the file is completely read, compressed and transferred to the remote server.

The software compression component in ADSM constructs a new compression dictionary for each file to be transferred. This compression dictionary is allocated and initialized at the start of each new file transfer and referenced throughout all the iterations of data chunk compression. When compression of the file is complete, a cleanup routine is called to deallocate the compression dictionary. Since the dictionary is generated from sequential processing of the file, parallelism within the compression operation can not be exploited in the current design of ADSM. The same is true for decompression.

In ADSM archival, each backup file has associated descriptor data, known as metadata, that describes attributes of the user's archived data. This metadata identifies that the archived data has been compressed by ADSM in software of hardware. If it has been compressed, subsequent data recovery operations will decompress the data based using the data encapsulation protocol to identify boundaries of each data original chunk of data. /footnote To insure robust operation, a software decompression algorithm that emulates the hardware compression algorithm was developed to provide uses with the ability to recover data without the use of hardware decompression.

ADSM currently uses a variant of the LZW [4] algorithm for software compression. LZW is based on dictionary based adaptive compression algorithms and is very efficient for software implementations. [3] As the encoder scans through the data file, all the unique substrings are entered into the dictionary. If the string occurs more than once, the second and latter occurrence will be encoded as the dictionary entry number, which has a more compact representation than the original string. Note that resultant compressed files don't have to carry the dictionary. The dictionary is represented implicitly. Entry numbers of the strings in the dictionary are in the order of appearance of these strings. The decoder can reconstruct the dictionary incrementally as it is decoding the compressed file. To improve performance, LZW optimizes compression speed by pruning the dictionary.

3 Implementation of Threaded Hardware Compression in ADSM

To provide hardware data compression within the ADSM product three alternative designs were considered. Namely, the "whole file method", the "Compression object replacement method" and the "data encapsulation method". The "whole file method" as the name denotes, compresses the entire file in hardware and then transfers it using the traditional ADSM stages with the software compression stage disabled. The second "compression object replacement method" has ADSM invoke to hardware compression objects in place of the legacy software objects when the card is detected. The "data encapsulation method", described below, overcomes limitations of the above methods with very little overhead.

The whole file method is, conceptually, the easiest to implement and support. Unfortunately, it does not match the existing architecture of ADSM for overlapping operations between Storage, CPU and Networks. This design serializes the hardware compression operation and forces data compression to complete before any other ADSM operation can begin. Additionally, compression of very large files requires large amounts of virtual memory resources to buffer and compress the entire file, at once, through hardware. We did not implement this approach for the above reasons.

The "compression object replacement method" is a straightforward way to include hardware compression into a system such as ADSM. The objective of this method is to replace the software compression routine by the LCS hardware compression. However, retaining the history buffer between data chunks requires additional coordination between the application and compression hardware since there is no direct way to reset all the data and control registers and keep the history buffer intact. This approach was prototyped but the CPU to PCI card coordination overhead was so large that performance improvements were limited.

[3]The LZW algorithm is used by many popular compression software such as the program "compress" in UNIX because of it's high efficiency for software compression.

	10 Mb/sec Ethernet			1000 Mb/sec Ethernet		
File Size	Software	Hardware	None	Software	Hardware	None
500K	1002-1620	719-1585	581-788	1091-1735	1219-2707	1253-2221
1M	1373-1823	1219-1843	756-870	1355-1855	2170-3497	1741-2266
2M	1908-2155	1617-1826	826-903	1711-2358	3260-3690	2033-2636
4M	2044-2385	2004-2148	876-937	2214-2520	3834-4551	2454-2767
8M	2320-2567	2085-2287	934-965	2276-2576	5017-5148	2690-2826
16M	2508-2652	2185-2268	927-970	2358-2708	4829-5360	2786-2837

Figure 1: Performance of ADSM Client/Server backup operation in Mbytes/sec with hardware, software and no compression.

Since the above approaches provided limited potential for performance improvements, we developed a novel "data encapsulation method" that supports threaded hardware parallelism within the existing ADSM architecture. The approach treats each chunk of data as an completely independent unit, or quantum, for compression. Since each quantum is independent, or stand alone, no compression history table state is required across quantum boundaries. This reduces the coordination between the PCI hardware and the ADSM application to a minimum. Furthermore, the difference in the compression ratio of a file with independent history tables within data chunks vs. whole file history is small. We have observed this difference to be less than 2% on average and never more than 5%. This is because the size of these data chunks are typically 32K bytes in size. This is sufficiently large enough to develop a useful history table and maintain a reasonable compression ratio over varying data types.

To separate and identify the compressed data chunks within an archived file, we added a data encapsulation protocol. This protocol consists of a header for each compressed buffer which, at minimum, contains the length of the compressed buffer. In this way, we have enabled multi-threaded compression with a penalty of 2% difference in compression ratio. Files of arbitrary size can be divided into many parts and handed to many threads, the compressed data from each thread are then encapsulated and put together to form the finished output file. Additionally, this allows the use of multiple hardware accelerators that increase performance through parallelism and data chunks within a file can be compressed and decompressed out of order, if the application so desires.

The hardware compression chip ALDC [3] uses the LZ1 algorithm. In the LZ1 algorithm, the encoder carries a history window which is the last N bytes that the encoder has seen. If the encoder sees a string that matches a substring in the history window, it will encode the string as a copy pointer referencing the substring in the history windows. A copy pointer has two fields, an offset between the two strings and the length of the strings, which provide enough information for the decoder to reconstruct the original string. [4]

4 Experimental Results

Our testbed implementation is based on Version 3.1 of the ADSM product release for Microsoft Windows NT and the updated client (for hardware compression) compiled using version 6.0 of Microsoft Visual C++ Universal Edition Compiler. The client and server machines are Pentium II 366 Mhz desktop workstations with 64K RAM and IDE disk drives running version 4.0 of Windows NT Workstation with Service Pack 4.

[4] It is worth noting that LZ1 is an asymmetric compression algorithm in that sense that decompression is much easier than compression. This is because while compression generally involves exhaustive search for matched substrings in the history windows, decompression requires only lookups. While software implementations of LZ1 are generally slow hardware implementations can be made very efficient by exploiting parallelism in Content Addressable Memory (CAM) for string matching.

Each machine is connected to a building supported 10 Mbps Ethernet as well as a dedicated point-to-point 1000 Mbps link. The client machine has associated drivers and the custom PCI card to support hardware compression when necessary.

To implement hardware based compression, we selected an efficient commodity compression chip offered by IBM Microelectronics. This chip, termed ALDC, performs data compression by scanning through the input buffer at the rate of one byte every clock cycle. Annapolis Micro System Inc. makes the Lossless Compression System board, (LCS), by mounting ALDC ICs to their WILD ONE (tm) boards. Two chips are clocked at 20 Mhz providing potential compression bandwidth of 40 MBps. The compression chips are interfaced to the ADSM client application through a custom API and device driver we previously defined in [1] provided with the LCS PCI Card.

Test data is made up of 7 files provided by IBM Microelectronics that exercise a range of compression ratios. Test data files include postscript files, jpeg images, GIF files and plain text documents. Although, each file varied in size from several kilobytes to megabytes, exact file sizes for analysis were created using concatenation and truncation of each file type. File sizes we considered were 500K bytes, and 1, 2, 4, 8, 16 Megabytes as shown in Figure 1. The range of performance for each experiment in Figure 1 shows the best and worst of archiving each of our collection of test files.

As described above, our testbed contained IDE storage disks which are slow (reduced cost to provide affordable computing) compared to a server with SCSI based storage. To eliminate the delay for disk access, we insure each file is resident is pre-fetched in memory prior to invocation of the ADSM backup operation. [5] Otherwise, we find that bandwidth is limited by the IDE disk drives and the only benefit of hardware compression is that CPU usage is reduced from 60to 10both bandwidth and CPU usage. The limiting factor for software compression with the file resident in memory is CPU usage.

Experiments we repeatedly run for client server backup for software compression, hardware compression and no compression for the same dataset. These we done using a 10 Mbps and 1000 Mbps Ethernet.

4.1 Software Compression vs. No Compression

The ADSM product allows the backup administrator to specify when to use compression. Our studies show that slow backup client/server network connections benefit from software compression and fast backup client/server networks do not. This is because, when the storage subsystem can support the bandwidth, the limiting factor in current systems is the slow network. Comparing the 10 Mbps Software vs. No Compression columns we find that the Mbyte/sec rates are nearly double for Software Compression. Note that the effective backup bandwidth of the software compression column exceed the peak rate of the network on all file size experiments. This is because software helped make the network more efficient in moving client/server backup data.

Removing the network bottleneck by using a point to point 1000 Mbps Ethernet connection, we find that software compression (CPU overhead) is now the bottleneck. In these columns, we find that no compression is more efficient than software compression. In all these experiments, the CPU loading was at 100% for all software based compression.

4.2 Software Compression vs. Hardware Compression

Examining the 10Mbps results for Software vs. Hardware compression we find that software actually outperforms the hardware based card in backup bandwidth. This is because the transfer is limited by the network bandwidth and not CPU resources. Since the software algorithm compress data at a higher ratio. This difference is due to the size of the compression history tables available in virtual memory vs. ALDC hardware. This means that every byte transferred across the fully loaded network is more meaningful than for the hardware compression implementation. Therefore, with less data to transfer across the fully loaded

[5]This is realistic to work with since Windows NT Server has a prefetch option which allows data to be cached in memory prior to use.

network, software compression appears more effective. Although, we find that hardware compression reduces the CPU loading from 100% to 40%.

Examining the 1000Mbps results for Software vs. Hardware compression we find that hardware outperforms software for all File Sizes and, once the CPU for software compression becomes fully loaded, we find the improvement grows to 100% in performance improvement. We also find that hardware performance improves as file size increases due to CPU loading.

4.3 Hardware Compression vs. No Compression

Comparison of the 10Mbps results for Hardware Compression vs. No Compression shows improvement over all File Sizes. The larger file sizes show an improvement of in excess of 100%. The 1000 Mbps results show that removing the network bottleneck allows for performance improvements of 18% to 89% from small to large files, respectively.

5 Application Benchmark Studies

Our testbed implementation for application benchmark characterization is based on Version 3.1 of the ADSM product release for Microsoft Windows NT and compiled using version 6.0 of Microsoft Visual C++ Universal Edition Compiler. The client and server machines are IBM Netfinity Servers Model 7000M10 running Microsoft Windows NT version 4.0 with Service Pack 4. The backup client includes up to four Intel Pentium III Xeon processors (450Mhz/1MByte L2 cache) with 4096MByte RAM, 18.2GByte of SCSI Disk Drives, one Alteon ACENic2 Gigabit Ethernet Card, and one Annapolis Micro Systems WildOne3 hardware-accelerated Lossless compression PCI card. Note that the Ethernet card was configured to use standard Ethernet packets (1518Bytes). The server includes up to four Intel Pentium III Xeon processors (450Mhz/1MByte L2 cache) with 2048MByte RAM, 27GByte of SCSI Disk Drives, configured in RAID0, and one Alteon ACENic2 Gigabit Ethernet Card. The client and server are connected with the dedicated gigabit Ethernet link.

Test data is made up of large blocks of archived data used for benchmarking of ADSM provided by IBM Storage Subsystem Division. This data consists of files consistent with a majority of customers using ADSM for backup and archival and the mix and distribution is proprietary information. Several directories were created in the client to hold 2.5 gigabytes of files with the following sizes, 256 Mbytes, 128 Mbytes, 64 Mbytes, 32 Mbytes, and a sampling of small files in the range of 16K bytes and below.

All resonable efforts were made to remove any bottlenecks that could restrict the flow of information from client to server. As described above striped SCSI disk arrays were used instead of a single IDE drive. The client and server were configured for single processors with 2048 Megabytes of memory available for each.

5.1 Backup Benchmark Execution Times

Figure 2 provides a table of results for the execution of the 2.5 Gbyte ADSM application backup benchmark in seconds. All times shown are for 2.5 Gigabytes of data transferred. Each row represents a different fix of file sizes from 10 separate 256 Megabyte files to 80 separate 32 Megabyte files. The Small row represents a mix of 1000's of small files in the range of 1 to 16 kilobytes in length for a total of 2.5 Gigabytes of data for backup as well.

We find that the Software Compression times for file sizes of 32M to 256M are roughly the same between 332 and 340 seconds. Using no compression offers an improvement of 7-8 percent over Software compression. This is because the network and striped storage subsystem has sufficient bandwidth to absorb the additional data across the link when the data is not compressed. Using the hardware method described, we reduced the time by over 58 percent for all large file benchmarks.

	Dedicated Backup Operation		
File Sizes	Software	Hardware	None
256M	340(99.94)	139(27.72)	319(14.27)
128M	334(99.95)	138(29.08)	313(13.39)
64M	337(99.79)	138(28.24)	336(13.64)
32M	332(99.72)	139(28.36)	345(15.69)
Small	569(60.80)	600(33.69)	569(26.71)

Figure 2: Time to execute the 2.5 Gigabyte ADSM Client/Server Backup Benchmark in Seconds with hardware, software and no compression. CPU percent utilization during the backup operation shown in ().

	Dedicated Restore Operation		
File Sizes	Software	Hardware	None
256M	298	222	196
32M	351	327	317
Small	515	486	449

Figure 3: Time to execute the 2.5 Gigabyte ADSM Client/Server Restore Benchmark in Seconds with hardware, software and no compression.

For small files, the overriding issue is the copying and management of the small files for transfer. These files are not large enough to offer significant reuse of string patterns to allow compression to help performance. Hardware compression is shown to be worse in this case by 7-8 percent due to the overhead of passing all data for compression through the kernel and the PCI card and back to the application for network transfers.

Figure 2 also provides utilization values for the duration of the data transfer in brackets. We find that large file software compression fully utilizes the CPU throughout the benchmark. This No Compression studies show that the application is CPU bound in performance for compression since time is not altered significantly but CPU usages is reduced by 83-85 percent. Using hardware compression we find that CPU utilization was reduced by 71-73 percent over Software compression. As expected, doing no compression has less CPU overhead.

5.2 Restore Benchmark Execution Times

Figure 2 provides a table of results for the execution of the 2.5 Gbyte ADSM application restore benchmark in seconds. All times shown are for 2.5 Gigabytes of data transferred. As before, each row represents a different fix of file sizes from 10 separate 256 Megabyte files to 80 separate 32 Megabyte files. The Small row represents a mix of 1000's of small files in the range of 1 to 16 kilobytes in length for a total of 2.5 Gigabytes of data for backup as well.

We see that software restore (Figure 3) is generally faster to do in the client server environment than backup (Figure 2). This is because, recovery or decompression of the files is a simple pointer look-up of previously used substrings. (The exhaustive search and matching is done on the compression.) This effect shows up in that hardware assisted restore is faster than software restore but not to the same degree. The percentage improvements range from 25 to 6 percent for the 32 M to 256M files as well as the small file sample set. Having backed up the data without compression offers the fastest restore times for all size files.

6 Conclusions and Recommendations for Future Work

Using high speed networks for ADSM backup and archival of data, software compression is a performance limiting bottleneck. It does not only slow down the compression, but it also imposes significant CPU overhead. Hardware compression speeds up the ADSM backup client, and off loads compression overhead from the CPU. We demonstrated that hardware compression improves the performance of ADSM by 2X to 3X while reducing the CPU loading by at least one-half, when using high bandwidth network communications. Previously, the CPU load was at 100% forcing the user's client offline for backup purposes.

Compression ratios can be improved for specific types of data if pre-processing filters are used prior to invoking hardware compression. These filters, as outlined in reference [2] can easily be placed in the programmable logic of our compression card. This would make their use transparent to ADSM except for the fact that additional compression is possible.

Additionally, an investigation into software pipelining of compression using ADSM data chunks is a promising avenue of research. This improved software design would emulate our novel data encapsulation method in software over multiple CPUs. Consider a 4, 8 or 16 way processor server that needs to compress a large file for backup archival purposes. One processor can call upon several idle CPUs to "independently" compress a chunk of data from the original file. This offers an advantage of speed over traditional, single history table, in line data compression.

The method described can also be applied to multi-threading software data compression across multiple CPUs in a shared or distributed memory parallel program environment. Furthermore, this methododology of threaded hardware acceleration can be applied to many other compute intensive algorithms, such as data encryption, that limit the performance of client/server systems.

7 Acknowledgments

Jain Yin facilitated the coding and integration of the LCS hardware into the ADSM product. Mazin Yousif of IBM Research Triangle Park provided the benchmark comparisons of the backup and restore functions. Ron Mraz provided the concept design, architecture and API specification for the LCS system for ADSM. Charles Nichols of IBM Tucson provided the benchmark data and procedures for the backup and restore operations. Bernard Lopez provided the testbed configuration and coordination of SSD and ARC activities. Jim Carlson is the IBM SSD point of contact. Special thanks to David Craft and Ted Lattrel of IBM Microelectronics who provided information and support of the IBM ALDC Compression Chip.

The authors would also like to thank David Pease, Jai Mennon, Ashok Chandra, Ken Vu, Paul Dantzig and Robert Morris who provided management support for this work. Finally, to Ron Arps who's thoughtful comments and insight helped formulate the quantum data methodology described in this report.

References

[1] Technical Reference:, "Lossless Compression System - Host Software Design" *Annapolis Micro Systems, Inc.*, 1999

[2] David J. Craft, "A fast hardware data compression algorithm and some algorithmic extensions" *IBM Journal of Research and Development*, Vol. 42, No. 6, 1998

[3] Technical Reference:, "ALDC" *ALDC1-20S-LP Data Sheet*, IBM Microelectronics Specifications, 1998

[4] Terry A. Welch, "LZW Compression", IEEE Computer, June 1984, pp. 8-19. *AIX Version 4.1 for RISC System 6000*, 1994

[5] Technical Reference:, “Tivoli Storage Manager Concepts” *IBM Redbook SG24-4877-02*, First Publish May, 2000