EFFICIENT END TO END DATA EXCHANGE USING CONFIGURABLE COMPRESSION

Yair Wiseman[1]
Georgia Institute of Technology
and Bar-Ilan University
wiseman@cs.biu.ac.il

Karsten Schwan
Georgia Institute of Technology
Patrick Widener
Georgia Institute of Technology
schwan,pmw@cc.gatech.edu

Abstract*-- *We explore the use of compression methods to improve the middleware-based exchange of information in interactive or collaborative distributed applications. In such applications, good compression factors must be accompanied by compression speeds suitable for the data transfer rates sustainable across network links. Our approach combines methods that continuously monitor current network and processor resources and assess compression effectiveness, with techniques that automatically choose suitable compression techniques. By integrating these techniques into middleware, there is little need for end user involvement, other than expressing the target rates of data transmission. The resulting network- and user-aware compression methods are evaluated experimentally across a range of network links and application data, the former ranging from low end links to homes, to wide-area Internet links, to high end links in intranets, the latter including both scientific (binary molecular dynamics data) and commercial (XML) data sets. Results attained demonstrate substantial improvements of this adaptive technique for data compression over non-adaptive approaches, where better compression methods are used when CPU loads are low and/or network links are slow, and where less effective and typically, faster compression techniques are used in high end network infrastructures.

Index terms-- **communication lines, compression**

I. INTRODUCTION

The amounts of data transported in modern grid applications can be substantial, often stressing even high performance communication infrastructures. For instance, in applications like DOE's Supernova Initiative[1], estimated data production rates by the simulations running on supercomputers exceed 1 GB/sec throughout a run. Similar data volumes are produced by remote sensors or instruments, such as earth observation satellites, specialized data sources (e.g., at nuclear research centers) and even in modern business applications like the operational information systems described in [2]. Coping with such data volumes typically requires users to devise and deploy application-specific methods to filter and select data, to ensure that the right data is received by the right end user at the right time[3,4].

In this paper, we explore the utility and limitations of using general, compression-based methods for reducing data volumes. In contrast to the lossy compression methods[5] used in previous work, for voice[6], image[7] and video[8] applications, this paper explores lossless compression techniques. This is because we are interested in integrating compression into the data exchange middleware used in today's grid computing infrastructures, where data loss would not be generally acceptable. The idea is to provide the levels of performance in data exchange end users require, with little or no involvement on their parts.

Our research focuses on interactive or collaborative, distributed applications, examples including remote collaboration via scientific or engineering data[4], remote visualization and graphics, and large-scale commercial codes like the operational information systems described in our previous research[2]. For such applications, as with the computational codes run across today's wide area grid infrastructures, compression methods cannot be applied blindly. Instead, their use must be dynamically configured to match current needs (e.g., desired transmission rates) with current platform resources (e.g., network bandwidth, CPU load). The techniques presented in this paper permits middleware to automatically configure compression. When sufficient network bandwidth is available, for instance, no compression is applied, thereby reducing computational loads. When network bandwidth is insufficient for the data volumes being exchanged, compression is applied. The specific compression method used is selected automatically, using dynamic data sampling techniques to assess the effectiveness and current speed of compression (since both are data-dependent). The intent is to ensure that the rate of data production (i.e., compression speed due to available CPU resources) and the effectiveness of compression (compression factor due to type of data) result in data volumes presented to the network at rates that match current network resources as well as application needs.

This paper demonstrates the utility of configurable compression in multiple execution environments. In one environment, relatively fast machines (e.g., high end PCs) communicate via low-end links, such as international Internet links across university collaborators (e.g., US to Israel) or DSL links connecting researchers' home machines with their institutions' computers. In another environment, workstation class machines are linked via the 1GB or 100MB links now commonly used in the Intranets of companies or research

[1] Corresponding Author. Address:
The Computer Science Department, Bar-Ilan University, Ramat-Gan 52900, Israel, Tel: 972-3-5317529, Fax: 972-3-5353325,
http://www.cs.biu.ac.il/~wiseman, wiseman@cs.biu.ac.il

institutions. Figure 1 depicts the heterogeneous computation/communication infrastructure used to realize these two environments.

The following are the results of our work. We first diagnose the effectiveness and performance of both standard and newer compression methods, using representative scientific and business data. The data sets used stem from a large-scale molecular dynamics simulation[4] and from the operational information system described in [2]. Given these data sets, the tradeoffs in compression speeds vs. reductions in required network bandwidth are assessed, in the two representative environments described in Figure 1. Second, based on these results, a table-driven, configurable approach to selecting suitable compression methods is developed. Method selection takes into account compression speed (dependent on current machine load), effectiveness (dependent on type of data), and available network bandwidth. Since platform resources change, method selection is performed repeatedly, throughout the lifetime of a data exchange between the parties the data exchange. Third, the resulting, configurable compression approach is integrated into middleware, resulting in a layered architecture that accommodates multiple compression methods, different network measurement techniques[9,10,11,12], and alternative communication protocols, including those well-suited for the large-data transfers[13].

Experimental evaluations indicate substantial performance advantages from using configurable compression, particularly for highly heterogeneous wide-area collaborations in which international collaborators[14] interact in real-time to jointly understand complex data sets or simulation results. For instance, we were able to significantly improve the speeds of data exchange for links from the U.S. to an Israeli university machine, in both low-load and high-load usage scenarios. Similarly, for home-based machines, even when using broadband links like DSL, notable performance advantages are attained from the use of compression. In Intranets, however, the utility of compression is less evident, especially in our own environments in which we have overcapacitated (i.e., relatively unloaded) networks offering from 100MB to 1GB connectivity between communicating machines. Generalizing from the testbed results described here, we expect configurable compression to compete well in embedded systems, as well, where they are best deployed on `tethered' machines before data is transmitted to mobile machines linked via wireless connections. Results with application-specific data reduction methods attained for embedded systems are described in [15].
In the remainder of the paper, we first briefly review the compression methods used in our work. In Section 2, we describe the various basic compression methods have been used by the configurable compression. Section 3 describes the software architecture of the IQ-ECho middleware into which configurable compression is being integrated. This middleware is targeted at large-data interactive applications, and it offers runtime mechanisms for online network bandwidth measurement and for managing the quality of middleware-based data exchanges (termed quality attributes), both of which will be used by the integrated version of configurable compression. The microbenchmarks discussed in Section 4 evaluate the basic performance characteristics of alternative compression methods, on different machines and with different data sets, to characterize their effectiveness vs. running time. A decision table and algorithm based on these results are used to dynamically and automatically select an appropriate compression technique for the data being streamed across distributed application components. Conclusions and future work appear in Section 5.

II. COMPRESSION METHODS

Compression methods reduce data size by applying compression and decompression techniques to data. This section briefly reviews the methods employed in our work, in order to permit the reader to better understand the tradeoffs in using these different techniques.

A. Huffman Compression

Huffman coding[16] - the first practical compression method - is used standalone[17] and within more complex compression techniques like JPEG[18,19]. The idea of Huffman coding is to assign a shorter codeword to a common item and a longer codeword to an uncommon item. Specifically, the following algorithm (shown in the recursive pseudo code below) finds codewords of minimum sizes for items $A_1,...,A_n$ of lengths $L_1,...,L_n$, where $P_1,...,P_n$ are the items' probabilities.

```
If (n==2) then {0,1}
Else
    Combine the 2 smallest probabilities Pn,Pn-1
    Solve for P1,P2,...,Pn-2,Pn-1+Pn
    If Pn-1+Pn is represented by α?then
        Pn-1 will be represented by α0
        Pn will be represented by α1
```

"Solve for" means calling the function again with n-1 elements (Pn-1 and Pn become one element, as indicated by $P_{n-1}+P_n$).

The main advantages of Huffman codes are their simplicity and speed. These codes work well for binary data when string repetition is rare. Huffman assumes that each character has no relation to the adjacent one; hence it usually does not perform well on texts. Huffman's complexity is O(m+nlogn) where m is the size of the text and n is the size of the alphabet. Note that this is much better than the Burrows-Wheeler compression technique described below, which has a complexity of O(mlogm).

B. Arithmetic Coding

Huffman coding uses a string of bits for each item in the original file. The arithmetic coding method[20,21] improves on this by using fractions of bits as codewords, where one bit can be `owned' by multiple items. As a result, a codeword can be represented by a rational numbers of bits (e.g., by 3.25 bits). The arithmetic coding method is described by the following pseudocode:

Let L be a set of items.
Every item $i \in L$ has a probability $P_i \in [0,1]$, such that:

$$\sum_{i \in L} P_i = 1$$

Every item is represented by the interval:

$$[\sum_{j<i} P_j, \sum_{j \le i} P_j)$$

- Repeat until EOF:
    - The current interval is divided into sub-intervals according to the items' probabilities.
    - Replace the current interval by the sub-interval of the items that were read.
- Write into the compressed file the shortest binary fraction available in the current interval.

C. Lempel-Ziv Methods

The traditional dictionary compression method is Lempel-Ziv coding[22,23]. WINZIP[24] and gzip[25] use versions of Lempel-Ziv coding. While Huffman coding and Arithmetic coding don't consider an item's environment, the main advantage of Lempel-Ziv methods is that they consider previous appearances of strings, as follows:

Let $x_1,...,x_n$ be a sequence of items.
We want to find a sub-sequence $x_k,...,x_m$ which holds:

$$P(x_k,...,x_m) > \prod_{i=k}^{m} P(x_i)$$

For example $p(qu) > p(q) \times p(u)$.

The Lempel-Ziv method puts a pointer into the place of each previewed string. We use a version of Lempel-Ziv that compresses these pointers by Huffman coding[26]. The pointers of Lempel-Ziv look like (100,7), which means go backward 100 bytes and copy 7 characters. Most pointers point to close destinations, and a copy of just a few bytes is done, so both of the numbers tend to be small. These numbers are represented by Huffman codes, which give shorter representation for small numbers.

D. The Burrows-Wheeler transformation

The Burrows-Wheeler transformation[27,28] is a dictionary compression method. The method utilizes repetitions of words' sequences in order to improve compression. No information is lost in the compression procedure. The procedure outperforms Lempel-Ziv coding, resulting in its use in a variety of compression utilities, but its execution time can be high. The method has multiple steps:

The first step creates pointers to all character of the file being compressed. The pointers are sorted according to the characters to which they are pointing. The preceding characters of each of the pointers are sent to the next step according to the order of the sorted pointers. Actually, this sequence of characters in the output has the same characters as in the original file, but the order of the characters is different.

The second step performs a "move to front" algorithm. This algorithm keeps all 256 possible characters in a list. When a character is to be sent to the next step, its position in the list will be sent. After the character is `sent', it is moved from its current position in the list to the front of the list (i.e., to position 0).

The next step applies a run-length coding to the output of the previous step. The output of the run-length coding is compressed by using either Huffman or Arithmetic coding.

The main disadvantage of the Burrows Wheeler transform is its slow execution speed, due to the need to sort the file. In order to reduce this speed, files are split into blocks, at some loss in compression factors, because shorter files are less effectively compressed. This paper uses the SGI version of the Burrows-Wheeler Transform[29].

In order to enable us to decompress the file when the order of blocks received does not exactly correspond to the order in which it is sent, we have adapted the Burrows-Wheeler method, as described next.

Each file is split into chucks of several lines. The Burrows-Wheeler Transform compresses each chunk. Then, chunks are processed by the move to front procedure, followed by run-length coding. The run-length coding is changed to use a run-length of at most 254 characters, so that the 255th character never appears. Instead, the 255th character is placed at the end of each compressed chunk. Next, all of the chunks are compressed jointly using Huffman coding. Huffman can be synchronized easily, as shown in [30]. This implies that if a Huffman-encoded file is read from any arbitrary point, it may have a few erroneous characters in the beginning, but the other characters will be correct. So, we can decode the compressed file from any arbitrary point, since Huffman will keep track of character positions, and when we see position 255, we have found the new chunk.

E. Method Comparison

We next assess the characteristics of these compression methods, so that it then becomes possible to choose the most appropriate one for any given characteristic of the data, the available communication bandwidth, and available CPU cycles. These characteristics are established in microbenchmarks described in Section 4.

Figure 1 qualitatively ranks compressions methods, scaled as:

- Excellent
- Good
- Satisfactory
- Poor

Given these method assessments, the following selection algorithm chooses the compression method most suitable for the current execution environment. In this algorithm, we use the term `reducing speed' to capture the speed at which (given currently available CPU cycles) a certain method is able to compress data. This speed is measured continually, as subsequent blocks of data are compressed. Also continually measured is the speed with which compressed blocks are accepted by receivers, thereby assessing both current network bandwidth and receiver speed. These end-to-end measurements are more relevant than knowledge of actual network bandwidth, since decompression requires the use of receivers' CPU cycles.

Assume the reducing size speed of first block is infinity.
While not EOF
Take a block of 128KB.
If (sending time) > 0.83*(the reducing size speed of Lempel-Ziv)
If sampling has been compressed into less than 48.78%
If (sending time) > 3.48*(the reducing size speed of Lempel-Ziv)
Use Burrows-Wheeler
Else
Use Lempel-Ziv
Else
Use Huffman
Else
Don't Compress
Fork a sampling process to compress the first 4KB of the next block by Lempel-Ziv and use its output to determine the reducing speed size and the compression ratio for the next 128KB block.
Send the block.
Wait for child process.

The sizes of the blocks have been chosen according to the efficiency of compression methods based on [31,32]. The ratios between the sending time and the reducing speed size have been set according to the statistics detailed in Figure 4. The efficiency of the sampling has been set according to the numbers of Figure 2. Obviously, this information is specific to the particular data used on section IV. However, these numbers can be tuned easily by sampling even a small piece of data[31] extracted from the original file and send this piece of data over an unloaded line employing unloaded CPUs. Usually, the numbers being used are very close to the constants details here, so we put these constants to give the reader an impression what the scope of the numbers is.

	Burrows-Wheeler	Lempel-Ziv	Arithmetic	Huffman
Compress files with string repetitions	Excellent	Excellent	Poor	Poor
Compress files with low entropy	Excellent	Poor	Excellent	Excellent
Compression Efficiency	**Excellent**	**Good**	**Poor**	**Poor**
Time of Compression	Poor	Satisfactory	Poor	Excellent
Time of Decompression	Satisfactory	Excellent	Poor	Excellent
Global Time	**Poor**	**Good**	**Poor**	**Excellent**

Figure 1

III. MIDDLEWARE INTEGRATION

It is unrealistic to expect end users to explicitly use the various compression techniques described in this paper. This section describes the mechanisms and architecture of middleware that permits end users to take advantage of configurable compression for a wide range of applications and platforms. The specific middleware used is ECho, an event-based communication system targeting large-data applications [33].

A. The ECho Middleware

Several attributes of ECho distinguish it from other middleware systems. First, ECho transports distributed data with performance similar to that achieved by systems like MPI. This allows it to support the large data flows on which our compression algorithms operate. Second, since ECho supports the publish/subscribe model of communication, new participants in event exchanges can be introduced by simply registering them with appropriate sets of events, without need for re-compilation or re-linking and without affecting other components, thereby facilitating system evolution. As improved compression algorithms are developed, or as ones more suitable for application needs are defined, this middleware capability allows applications to take advantage of such methods without any associated re-engineering costs. Third, ECho supports the definition and use of globally named and interpreted quality attributes. Using attributes, ECho can transport performance information and/or dynamic change instructions, across end users and address spaces and across different implementation layers, including the application level,

the operating system kernel, and the network level. This functionality is useful with online compression, since attributes can transport information about network or processor state to compression methods and/or to the algorithm that selects the compression methods to be used.

Event subscription utilizes event channels, which are the mechanisms through which event producers and consumers are matched. Producers submit events to a specific channel, and only consumers subscribed to that channel are notified of those events. Conceptually, event channels exist in the space `between' processes, but in practice they are distributed entities, with bookkeeping shared between all processes where they are referenced. Since channel implementations are distributed, it is straightforward for ECho to apply computations - termed handlers - to events, at any point in the data path between event producer and consumer. Handlers may transform events, reduce their sizes or enhance the information they contain, and they can even prevent events from being transported, if needed. They are the key to the integration of compression methods, discussed next.

B. Integrating Compression Methods

Since compression methods are integrated into ECho as event handlers, they can control the amount of data sent by producers and thereby also control the network bandwidth used by ECho applications. Moreover, with handlers as execution vessels for data compression, a new compression method can be introduced at any time during a system's operation. In ECho, such dynamic handler instantiation is known as deriving a new event channel from an existing one; it is an action taken by event consumers (event producers cannot take the responsibility of customizing event delivery for all or some subset of their consumers since event channel subscription is anonymous).

Section 2.5 describes a decision process, which selects a compression method that will perform best in the current execution environment. This decision method is integrated into ECho middleware by adding it the actions taken at event consumers. When the method determines that a consumer should change its compression method, it informs the appropriate source of this change via attributes. If the source does not yet have the suitable compression method, the consumer deploys a new method by simply deriving the appropriate event channel with that method. Having done so, the consumer can then unsubscribe from the original channel and subscribe to the new one, thereby connecting to an event stream with newly embedded data compression. Since the consumer selected the specific new data compression method, it knows which decompression method to apply to any received event to correctly reconstruct the application data. ECho channels utilize a transport encapsulation layer that efficiently multiplexes multiple connections from a single address space, so maintaining a small number of open channels and switching among them as the operating environment changes does not adversely affect performance.

C. Other Middleware Approaches

While we have integrated compression methods into the ECho middleware, the approach of dynamically changing compression algorithms and degrees of compression in response to execution conditions is equally valid with other systems. For example, middleware built on top of the Java RMI package could introduce these computations as Java code in the classes implementing communication. Similarly, in .NET, compression could be integrated into its `remoting' methods. CORBA-based systems could deploy compression codes as Interceptor modules, which are placed just before data is transmitted from the sender and just after data is received at the receiver. The ECho middleware system provides higher performance than either RMI or CORBA-based systems, but the overall integration approaches are similar.

IV. EXPERIMENTAL RESULTS

A. Microbenchmarks.

The compression methods used in our work (i.e. Huffman, Arithmetic, Lempel-Ziv and Burrows-Wheeler) have been tested with multiple datasets, including scientific data and a dataset of a large commercial company. The binary data sets used are represented in an efficient format developed by our group, termed PBIO[34]. For the commercial data, the compression ratios achieved are shown in Figure 2.

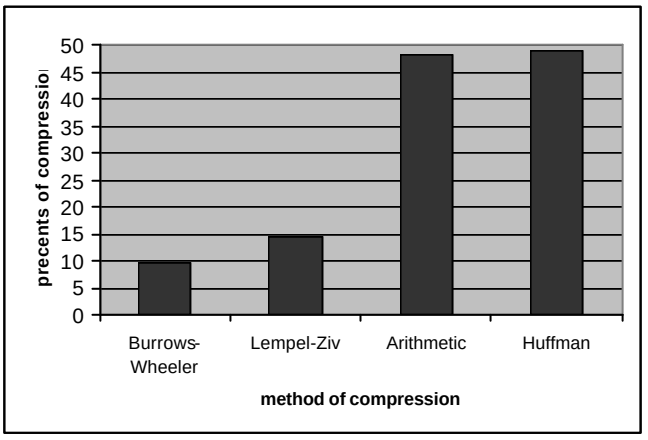

Figure 2

The corresponding times of compression and decompression appear in Figure 3. These times were measured on a Sun-Fire-280R with UltraSPARC-III processor running Solaris 5.8. The machine had 4GB of internal memory.

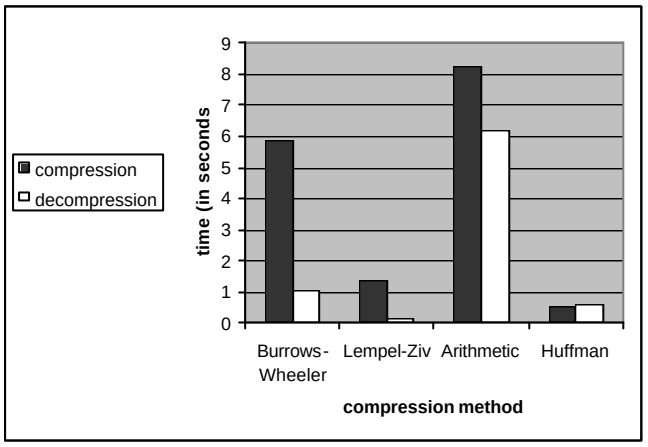

Figure 3

The key insight from these experiments is the speed with which a CPU compresses some large amount of data. Figure 4 summarizes the test results attained with two Sun machines, one being the same Sun-fire as the one in Figure 3, the other an Ultra-Sparc machine with an UltraSPARC-II processor running Solaris 5.8 and having 128MB of main memory.

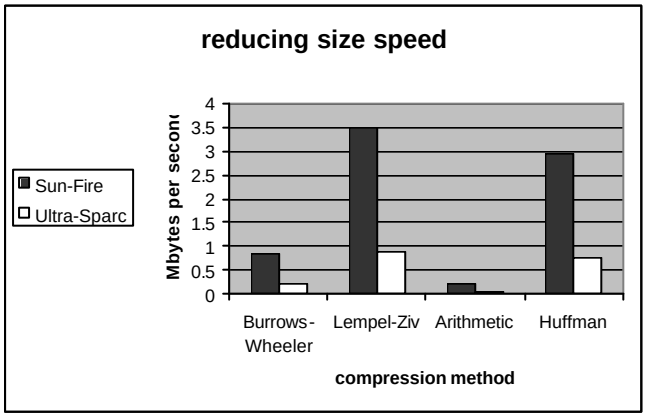

Figure 4

Figure 4 depicts what we term the `reducing speed' of different CPU, which is the number of bytes per second by which a CPU can reduce data. If such space reduction can be performed faster than the transfer time for a given amount of data, it is worth (time-wise) to compress the data. If the CPU is fast, but the communication line is slow, even a complicated compression method can be chosen. In the reverse case, i.e. the CPU is slow and the communication line is fast, no compression method will be assigned. Between those two extremes, the use of a fast and uncomplicated compression method is indicated.

Experimental results were attained with 1GB/s, 100MB/s, and 1MB/s links. We also used an international Internet link between the Georgia Institute of Technology, Atlanta, GA and Bar-Ilan University, Ramat-Gan, Israel. Figure 5 shows the speed of these communication lines. All of the measurements were taken on warm lines. The standard deviation of the transfer speed were 0.782%, 8.95%, 1.17% and 46.02% on the 1GB, 100MB, 1MB and the international links, respectively.

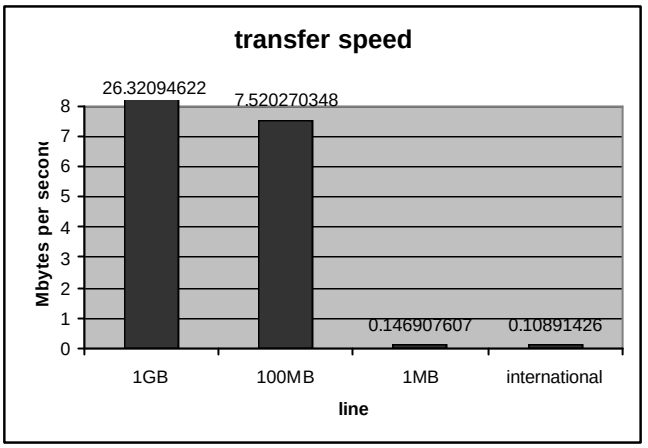

Figure 5

The conclusion from the above figures is that if the CPU is very fast, then Burrows-Wheeler is the best method. Burrows-Wheeler has a bad ratio of reducing MBs per second, but if the CPU is fast enough, the CPU can pay back any lost time. If the communication line is very fast, Huffman will be the best method. Huffman has a bad compression ratio, but compression is quite fast. When an intermediate case is to be used, Lempel-ZIV realizes a good compromise between compression time and transfer time. Arithmetic coding does not appear useful for the class of applications considered in our work. On a local fast communication link (e.g., a 1GB link) or on a lightly loaded 100MB link, compression appears too expensive in general. This suggests that compression should not be used at all.

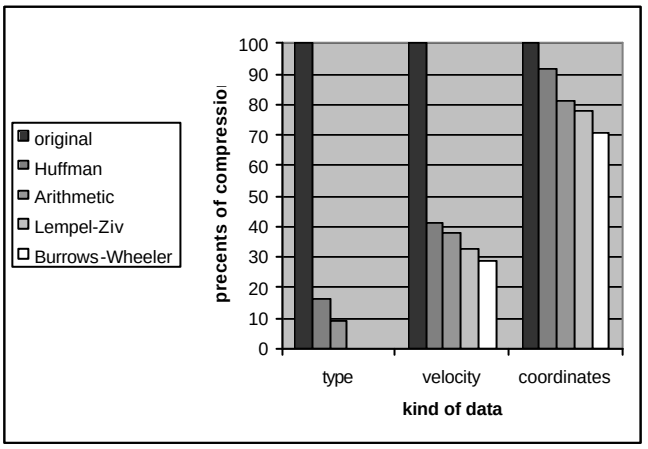

Figure 6

We tried our methods on scientific data sets. The specific data used is from a molecular dynamics application; it contains the coordinates of atoms, their velocities and their types. The compression ratios of the different data items are quite different from each other, as can be seen in Figure 6. However,

the compression times across these data sets do not differ much.

The conclusion from Figure 6 is that decisions about suitable compression techniques should be based not only on data sizes or link speeds, but also on data characteristics. Huffman codes and Arithmetic codes are suitable for low entropy data, while Lempel-Ziv methods are good at handling data with string repetitions. Burrows-Wheeler handles both of these cases.

The consequent approach taken in our work is one that samples data as it is being produced and transported, to detect whether data has low entropy, string repetitions, or both. The results of such sampling are used to choose a suitable compression method. With ECho, such sampling can be integrated into the producer-side actions taken on events, much like we are currently integrating network measurement methods into the enhanced IQ-ECho implementation targeted at Internet links[13].

B. Experimentation with applications.

In order to evaluate the behavior of compression with realistic scenarios, we created an environment in which network load is varied artificially, using load traces captured for the MBone multicast infrastructure. These commonly used traces track the number of end users that connect to MBone sessions over time, thereby indirectly describing the network load implied by MBone use [35]. Consequently, in Figure 7, load is stated as the number of connections over time.

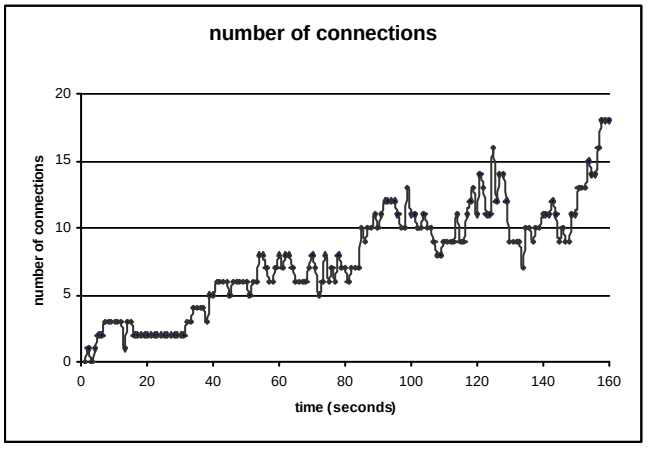

Figure 7

The experiments presented below employ the MBone trace by creating network loads with it, using the raw MBone numbers multiplied by a factor of 4 in order to adjust it to the capacities of the 100MB links used in our experimentation. The data being compressed, transported, and decompressed is a set of transactions captured from the operational information system of a large company with whom we are working. This data set has a high rate of strings repetitions, so the best methods to be used were Lempel-Ziv and Burrows-Wheeler. These comments explain the automatic decision-making depicted in Figure 8 (experiments are conducted on the same machine as the one used in Figure 3). Initially, with no network load, no compression is performed (labeled as `1' in the figure). With increasing network load, the first compression method used is Lempel-Ziv (see `2' in the figure), followed by Burrows-Wheeler (see `3') under high network loads.

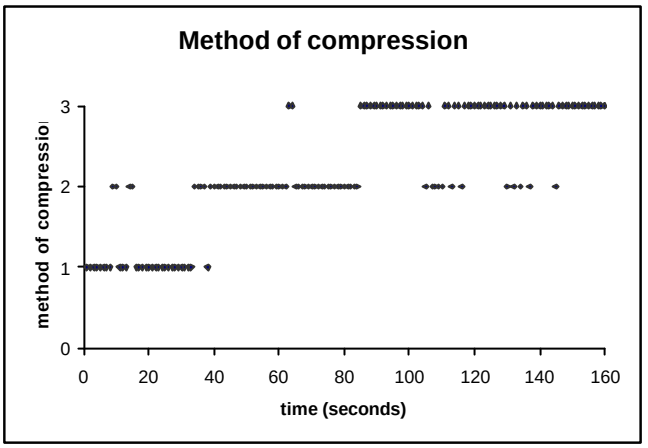

Figure 8

Figure 8 depicts the choices of compression method over time, given the MBone-based network load. Figures 9 and 10 depict the compression times and the sizes of the compressed blocks, respectively, attained by these methods. From these figures, it is clear that the relatively small gains in data reduction attained from the use of Burrows-Wheeler justify its use only under very high network loads.

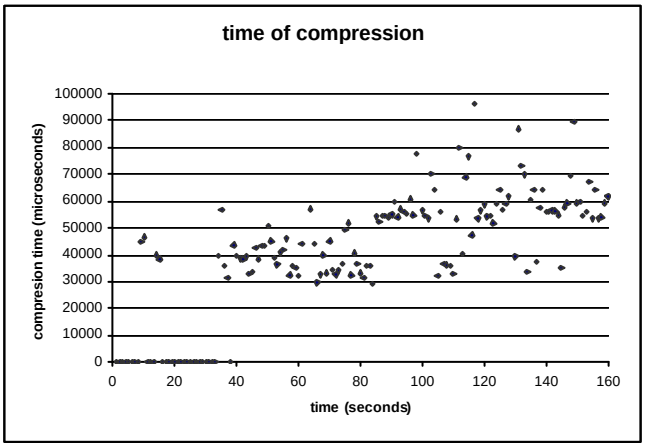

Figure 9

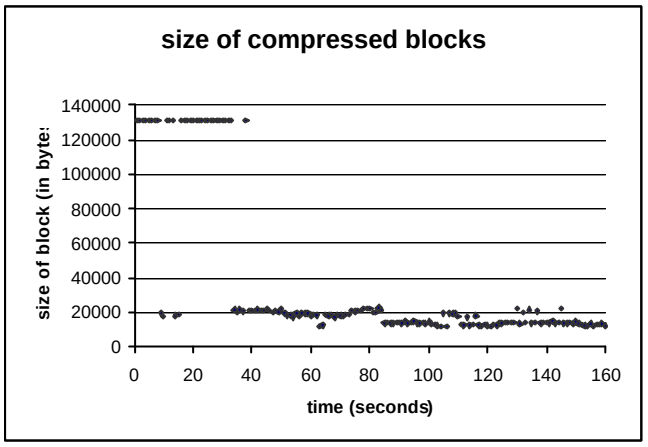

Figure 10

The next set of experiments is based on the same network load behavior, but uses the molecular data introduced earlier. In particular, our microbenchmarks (see Figure 6) show that the data containing atom coordinates does not compress very well. When applying configurable compression to this data set, the results depicted in Figures 11 and 12 are attained. These results are explained next.

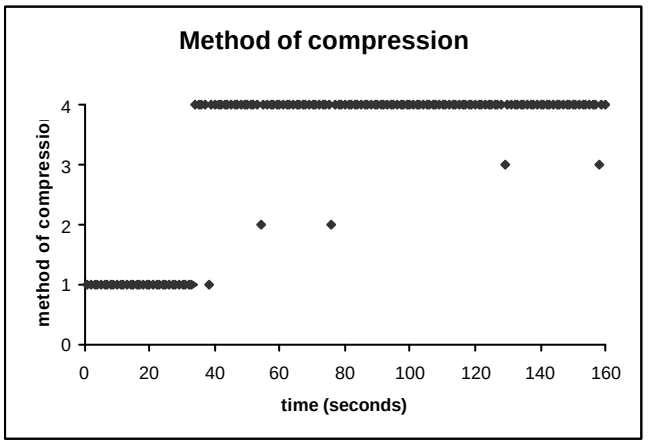

Figure 11

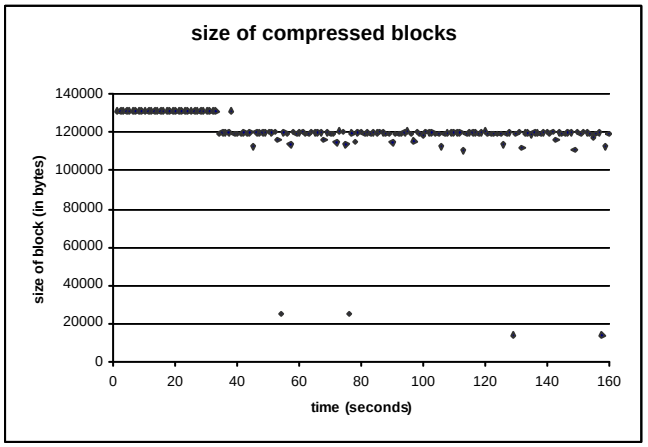

Figure 12

Most of the data cannot be compressed well, as was shown in Figure 6. However, there are some small portions of the data that have strings repetitions. Those portions are recognized by configurable compression, which applies Lempel-Ziv (to the string portions) or Burrows-Wheeler (to the other data) in order to improve the compression ratio. As can be seen in Figure 11, most of the data was compressed by Huffman. In Figure 11, `1' indicates no compression, while `2', `3' and `4' indicate Lempel-Ziv, Burrows-Wheeler, and Huffman compression, respectively.

V. CONCLUSIONS AND FUTURE WORK

This paper evaluates multiple general, lossless compression methods, in terms of their effectiveness of use in distributed transactional or streaming data applications, in heterogeneous distributed systems and for datasets that include both commercial transactional and scientific data. The paper also describes how to integrate compression into modern middleware, thereby removing from end users the potentially onerous task of choosing suitable compression methods. This is important because in typical distributed systems and applications, the choice of compression should be dynamic, depending on currently available network bandwidth, computational load on end user machines, and the kind of data (i.e., its compressability) being transported.

Initial results are encouraging. The notion of `quality attributes' in middleware enables the end-to-end and cross-layer interactions needed to transport the monitoring and configuration data across systems that is needed (1) for sampling the data being streamed and (2) for continuously capturing network and machine loads, thereby deciding upon the suitable compression method to be used for middleware-level data transmission. (3) By permitting end users to dynamically change the parameters used by compression methods, they can also explicitly affect compression behavior. Most interestingly, (4) end users can also `derive' from an existing middleware-based interaction a new interaction, new compression methods can be deployed into systems at runtime, permitting them to take advantage of new methods or of methods more suited to their application domains and data sets.

Configurable compression was evaluated with both scientific and commercial data sets. Using configurable compression, we could transport the transactional data of a large company with which we are working on a 100MB network link under variable load in 10.7142 seconds (where compression took slightly more than 60% of total time) rather than in the 29.1388 seconds it took without compression.

In comparison to the commercial data set, some of our scientific data is not easily compressed. Specifically, the random characteristic of that data requires us to use the

`strongest' compression methods available, such as Burrows-Wheeler. Due to the overheads of using such methods, dynamic data compression actually increases the total time required for data streaming, from roughly 29 to 30.5 seconds. Cases like these indicate the importance of permitting end users to integrate their own, application-specific, lossy compression techniques into data streaming middleware. This is a topic of our current work, to make it easy and efficient for middleware to execute high performance compression methods and to instrument such methods to vary their dynamic behavior based on current platform resources[4].

References

[1] DOE-TSI. Terascale supernova initiative. http://www.phy.ornl.gov/tsi.

[2] **V. Oleson, K. Schwan, D. Amin, G. Eisenhauer, B. Plale, C. Pu, and P. Widener**, Operational Information Systems, An Example from the Airline Industry, *Workshop on Industrial Experiences with System Software (WIESS 2000), in conjunction with OSDI 2000*, November 2000.

.

[3] **B. Plale, G. Eisenhauer, K. Schwan, J. Heiner, V. Martin, and J. Vetter.** From interactive applications to distributed laboratories. *IEEE Concurrency, 6(3),* 1998.

[4] **M. Wolf, Z. Cai, W. Huang, and K. Schwan**. Smart Pointers: Personalized Scientific Data Portals in Your Hand, *In Proc. of Supercommputing 2002*, Nov. 2002.

[5] **M. Beigl**, MODBC - A Middleware for Accessing Databases from Mobile Computers. 3rd Cabernet Plenary Workshop, Rennes, France, 1997

[6] MMEV voice compression/decompression middleware, http://www.asahikasei.co.jp/vorero/en/onsei2/mmev1.html.

[7] INNOTECH Corporation, IMPress Version 2.0, Image compression and decompression middleware system for embedded systems. http://www.innotech.co.jp/english/news/contents/news021029e.html.

[8] Envivio Corporation, H.264 Live Solution for Cost-Effective Delivery of Broadcast-Quality Video Over Satellite Networks In Proc. of NAB 2003, Apr. 2003.

[9] **M. Mathis.** Web100 and the End-to-End problem. http://www.web100.org/docs/jtech/.

[10] **N. S. Rao, Y.-C. Bang, S. Radhakrisnan, Q. Wu, S. S. Iyengar, and H. Choo**. NetLets: measurement-based.

[11] **M. Jain and C. Dovrolis.** End-to-end available Bandwidth: Measurement methodology, Dynamics, and Relation with TCP throughput. *In Proceedings of ACM SIGCOMM*, Aug. 2002.

[12] **M.Jain and C. Dovrolis.** End-to-End Available Bandwidth: Measurement methodology, Dynamics, and Relation with TCP Throughput. *IEEE/ACM Transactions in Networking*, August, 2003.

[13] **Q. He and K. Schwan.** IQ-RUDP: Coordinating Application Aaptation with Network Transport. *In High Performance Distributed Computing*, July 2002.

[14] GriPhyN. The grid physics network. http://www.griphyn.org.

[15] **Y. Chen, K. Schwan, and D. Zhou**. Opportunistic channels: Mobility-aware event delivery. *In Proceedings of the ACM/USENIX International Middleware Conference*, 2003.

[16] **Huffman D.** A method for the Construction of Minimum Redundancy Codes *Proc. of the IRE 40, pp.1098-1101* 1952.

[17] **Bookstein A., Klein S.T.,** Is Huffman coding dead?, Journal of Computing 50, pp. 279-296, 1993.

[18] **Wallace G. K.** *The JPEG Still Picture Compression Standard* Communication of the ACM 34, pp. 3-44, 1991.

[19] Information Technology *Digital Compression and Coding of Continuous-Tone Still Images Requirements and Guidelines* International Standard ISO/IEC 10918-1, 1993.

[20] **Witten I. H., Neal R. M. and Cleary J. G.** Arithmetic Coding for Data Compression, *Communication of the ACM 30, pp. 520-540* 1987.

[21] **Howard P. G. and Vitter J. S.,** Arithmetic Coding for Data Compression, *Proceedings of the IEEE, 82(6),* pp. 857-865, 1994.

[22] **Ziv J., Lempel A.**, A universal algorithm for sequential data compression, *IEEE Transactions on Information Theory, IT-23*, pp. 337-343, 1977.

[23] **Ziv J., Lempel A.**, Compression of individual sequences via variable-rate coding, *IEEE Transactions Information Theory, IT-24*, pp. 530-536, 1978.

[24] WinZip, *Nico Mak Computing, Inc., Mansfield, CT, USA* 1998.

[25] gzip, *Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA, USA* 1991

[26] Brent. R. P., *A linear algorithm for data compression*. Australian Computer Journal, 19(2) pp. 64-68, May 1987.

[27] **Burrows M. and Wheeler D.** Block sorting Lossless Data Compression Algorithm, *System research center, research report 124, Digital System research Center, Palo Alto, CA* 1994.

[28] **Nelson M. R.** Data Compression with the Burrows Wheeler Transformation, *Dr. Dobb's Journal, pp. 46-50* 1996.

[29] SGI® IRIX® Freeware distribution, 1600 Amphitheatre Pkwy. Mountain View, CA, USA, Edition of February 2003.

[30] **Klein S. T. and Wiseman Y.** Parallel Huffman Decoding with Applications to JPEG Files, *The Computer Journal, Swindon,* UK, 2003.

[31] **Wiseman Y.,** Parallel Compression, Ph.D. Thesis, Bar-Ilan University, Ramat-Gan, Israel, 2000.

[32] **Klein S. T. and Wiseman Y.,** Parallel Lempel Ziv Coding, *The Journal of Discrete Applied Mathematics*, 2003.

[33] **Eisenhauer G. and Schwwan K., The ECho Event Delivery System**, *Technical Report GIT-CC-99-08, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280*, 1999.

[34] **Plale B., Eisenhauer G., Daley L. K., Widener P. and Schwan K.,** Fast Heterogenous Binary Data Interchange for Event-based Monitoring, *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS2000),* 2000.

[35] **Cai Z., He Q., Eisenhauer G., Schwan K. and Wolf M.,** IQ-services: Network-Aware Middleware for Interactive Large-Data Application , *submitted to Supercomputing (SC2003),* May 2003.