Fast Lossless Compression of Scientific Floating-Point Data

Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher
Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University, Ithaca, NY 14853
{paruj, jke, burtscher}@csl.cornell.edu

Abstract

In scientific computing environments, large amounts of floating-point data often need to be transferred between computers as well as to and from storage devices. Compression can reduce the number of bits that need to be transferred and stored. However, the run-time overhead due to compression may be undesirable in high-performance settings where short communication latencies and high bandwidths are essential. This paper describes and evaluates a new compression algorithm that is tailored to such environments. It typically compresses numeric floating-point values better and faster than other algorithms do. On our data sets, it achieves compression ratios between 1.2 and 4.2 as well as compression and decompression throughputs between 2.8 and 5.9 million 64-bit double-precision numbers per second on a 3GHz Pentium 4 machine.

1. Introduction

Some environments require the fast transfer of large amounts of numerical data over a network. Examples include parallel scientific programs that exchange intermediate results between compute nodes after each simulation step, the storage and retrieval of numeric data sets to and from servers, and sending simulation results to other computers for further processing, analysis, visualization, etc. While compressing such data reduces the number of bits that need to be transferred and stored, the compression/decompression overhead typically increases the communication latency and decreases the effective bandwidth, which is undesirable in high-performance computing environments.

This paper describes a lossless compression algorithm for 64-bit floating-point values that is fast enough to support software-based real-time compression and decompression in settings such as the ones described above. In many instances, its compression and decompression overhead is lower than the time saved by transmitting the shorter, compressed data. For example, we have included our algorithm in a message-passing interface (MPI) library [8]. Parallel programs use such libraries to exchange data between the CPUs in a parallel system. Unbeknownst to these programs, our library automatically compresses the message data before they are sent and decompresses them at the receiver. In this fashion, we have sped up scientific numeric programs by 3% to 98% on the 128-node Velocity+ cluster [23].

The algorithm we present in this paper was inspired by VPC [4], a value-prediction-based compression algorithm for program execution traces. In recent years, hardware-based value predictors [11], [28], [31] have been researched extensively to accurately

predict the content of CPU registers, including floating-point registers. These predictors use simple and fast algorithms because they are designed to make billions of predictions per second, which makes them good models for our compression purposes.

Our algorithm compresses sequences of IEEE double-precision floating-point values as follows. First, it predicts each value in the sequence and XORs it with the true value. If the predicted value is close to the true value, the sign, the exponent, and the first few mantissa bits will be the same. XOR is a reversible operation that turns identical bits into zeros. Since the sign, the exponent, and the top mantissa bits occupy the most significant bit positions in the IEEE 754 standard (Figure 1), we expect the XOR result to have a substantial number of leading zeros. Hence, it can be encoded and compressed by a leading zero count that is followed by the remaining bits.

Figure 1: Double-precision floating-point format of the IEEE 754 standard

The prediction operation is a fast hash-table lookup. XORing two values and counting the leading zeros are also fast operations. As a consequence, both compression and decompression are very quick with our algorithm.

Unlike most related work on compressing floating-point data (Section 3), our approach does not rely on consecutive values differing by only a small amount. Rather, it tries to identify recurring difference patterns. In particular, for every processed subsequence of *n* differences between consecutive values (i.e., the context), it records the next difference in a hash table. When making a prediction, a table lookup is performed to retrieve the difference that followed the last time a similar sequence of *n* differences (i.e., a similar context) was observed. By similar we mean that the top *m* bits of the floating-point values are the same. The predicted difference is then added to the previous value to form the final prediction. This approach works well in cases where subsequent data correlate reasonably with earlier data, which is often the case for the intermediate and final floating-point results produced by scientific programs.

The rest of this paper is organized as follows. Section 2 explains our compression algorithm in more detail. Section 3 discusses related work. Section 4 describes the evaluation methodology. Section 5 presents the results. Section 6 summarizes the paper.

2. Compression algorithm

Our algorithm is designed for compressing arrays of 64-bit IEEE double-precision floating-point values. It employs a predictor to forecast the next array element based on earlier elements. To compress a double, it encodes the difference (XOR result) between the predicted and the true value. If the prediction is close to the true value, the difference can be encoded with just a few bits.

Figure 2 illustrates how the 236th entry D_{236} in an array of doubles is compressed. First, the predictor (Section 2.1) produces a prediction *Pred*. Then we XOR D_{236} and *Pred* to obtain the difference *Diff*. *Diff* has many leading zero bits if *Pred* is close to D_{236}. The leading zeros are then encoded using a leading zero count (LZC_{236}). The remaining bits ($Bits_{236}$) are not compressed.

We use four bits for the leading-zero counts, which encode *4*LZC* zeros. This choice allows us to work at half-byte granularity, which is more efficient than working at bit granularity. Note that this scheme provides the same average code length as a six-bit leading-zero count if the counts are evenly distributed. We do not use a more sophisticated compression scheme to keep the compression and decompression time small.

Many CPUs, including Intel's Pentiums [20], support machine instructions for counting leading zeros, which can be used to accelerate our algorithm. However, to make the performance comparisons fair, we only show results obtained with a portable C implementation of our algorithm in this paper.

During decompression, our algorithm first reads the four-bit leading-zero count *LZC* and then *64-4*LZC* effective bits to regenerate the difference *Diff*. The predictor in the decompressor is kept consistent with the compressor's predictor by updating them with the same values, i.e., the previously seen doubles. Thus both predictors are guaranteed to produce the same prediction *Pred*. The true value D_{236} can therefore trivially be regenerated by XORing *Diff* with *Pred*.

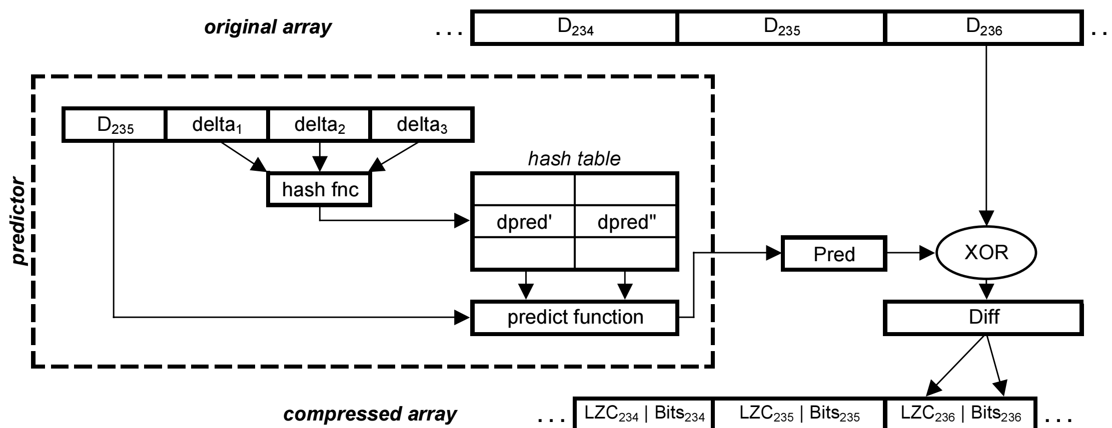

Figure 2: Compression algorithm overview

2.1. Predictor

We use a differential-finite-context-method predictor (DFCM) [11] in our compression algorithm. It computes a hash out of the *n* most recently encountered differences (integer subtraction) between consecutive values in the original array, where *n* is referred to as the order of the predictor. Figure 2 shows the modified third-order DFCM predictor we use. It performs a table lookup using the hash as an index to retrieve the differences that followed the last two times the same hash was encountered, i.e., the last two times that same sequence of last three differences was observed. The retrieved differences are used to predict the next value by adding them to the previous value in the array as explained below. Once a prediction has been made, the predictor is updated with the true difference and value. We modified the predictor as follows to work well on floating-point data.

Hash function: For sequences of floating-point values, the chance of an exact 64-bit prediction match is very low. Moreover, it is desirable that, for example, the decimal difference sequence <0.5001, 0.6001, 0.9001> be hashed to the same index as the sequence <0.5000, 0.6000, 0.9000>. For these reasons, our hash function uses only the *m* most sig-

nificant bits and ignores the remaining bits. Our experiments show that hashing the first fourteen bits (the sign bit, the eleven exponent bits, and the top two mantissa bits) results in the best average prediction accuracy. We use the following hash function:

$$\text{hash}(delta_1, delta_2, delta_3) = \text{lsb}_{0..19}(delta_1 \otimes (delta_2 << 5) \otimes (delta_3 << 10))$$

In this function, $\otimes$ denotes bit-wise XOR, $<<$ represents bit-wise left shift with zero insertion, and the $delta_i$ stand for the fourteen most significant bits of the three most recent differences between consecutive array elements. The lowest five bits of each $delta_i$ hold the three least-significant exponent and the two mantissa bits and thus contain the most frequently changing bits. Shifting the deltas by five bits relative to each other before XORing them moves these frequently changing bits into non-overlapping positions, which we found to decrease detrimental aliasing in the hash table. We only use the twenty least-significant bits of the XOR result for the index because our hash table has 2^{20} (about one million) lines. Larger hash tables slow down the algorithm and do not increase the prediction accuracy noticeably.

Prediction function: Note that we keep the most and the second-most recent value in each line of the hash-table (*dpred'* and *dpred"*). Both values are full 64-bit differences. The predictor uses *dpred'* if *dpred"* and *dpred'* are not close to each other, i.e., their first fourteen bits are not the same. Otherwise, the predictor uses *dpred'*+(*dpred'*-*dpred"*), where the *dpred'*-*dpred"* term accounts for the drift in the difference values, which we found to improve the prediction accuracy and thus the compression ratio.

3. Related work

Most previous work on lossless compression of floating-point values focuses on data from audio, image, scientific measurement, and simulation domains. The majority of such data is represented in the 32-bit IEEE 754 single-precision format. In contrast, our work focuses on the 64-bit double-precision values produced by numeric programs.

Klimenko et al. [27] present a method that combines differentiation and zero suppress algorithms to compress floating-point data arising from experiments conducted at the Laser Interferometer Gravitation Wave Observatory (LIGO E2 data). It has about the same compression ratio as gzip but is significantly faster. Its success is tied to the nature of the LIGO data, which are time-series whose values change only gradually.

A paper by Engelson et al. [7] is the only work we are aware of that proposes a scheme for compressing 64-bit floating-point values. Their data comes from the output of a numerical solver for ordinary differential equations. The authors use integer delta and extrapolation algorithms to compress and decompress the data. Similar to the work by Klimenko et al., the success of this method depends on the "smoothness" of the data, i.e., the difference between consecutive values is typically small and can therefore be encoded with only a few bits.

Several papers [9], [25], [33], [34] concentrate on compressing floating-point data that represent images. These studies focus on maximizing the compression ratio as the (de)compression speed is not so relevant. Usevitch [34] proposes extensions to the JPEG2000 standard that allow floating-point data to be efficiently encoded with bit-plane coding algorithms where the floating-point values are represented as "big integers". Gamito et al. [9] describe modifications needed in JPEG2000 to accommodate lossless floating-point compression, namely, adjustments in the wavelet transformation and ear-

lier signaling of special numbers such as NaNs in the main header. Isenburg et al. [25] employ an arithmetic coder for single-precision floating point fields that represent residual vectors between the actual and the predicted vertex positions in triangular meshes. Trott et al. [33] use an extended precision digits algorithm, the Haar wavelet transform, and Huffman coding to losslessly compress 3D curvilinear grids.

Ghido [10] proposes an algorithm for lossless compression of floating-point audio data. It transforms the floating-point values into integers, producing quantized sequences, and generates an additional binary stream used for the lossless reconstruction of the original floating-point values.

Utilizing value predictors as data models for compression purposes has previously been explored for program-execution-trace compression [3], [4], [5]. However, that work employs multiple predictors, which we cannot afford in this work because of the tight time constraints. Another paper by the authors [26] describes how we included the compression algorithm described in this paper in an MPI library to speed up parallel message passing programs running on a cluster of workstations.

4. Evaluation methodology

4.1. System

We performed all measurements for this study on a 3GHz Intel P4-Xeon system with 1GB of main memory. The hard disk is a Seagate Cheetah 10K.6 Ultra320 SCSI, with a capacity of 37GB, 8MB of build-in cache, and a spin rate of 10,000RPM. The operating system is Linux Suse 9.1. Except where executables are directly available for our platform, the compressors used in this study, including ours, were compiled with the GNU C compiler (gcc) version 3.3.3 with the -O3 optimization flag.

4.2. Timing measurements

All timing measurements in this paper refer to the sum of the user and the system time as reported by the UNIX shell's *time* command. In other words, we report the CPU time and ignore any idle time such as waiting for disk operations. All tested algorithms read data from the hard disk and write data back to the hard disk. Given the large sizes of our data sets, any effects due to disk caching should be minimal.

4.3. Other compressors

We compare our algorithm to six general-purpose and one special-purpose compressor. This section briefly introduces these compressors.

We implemented the fsd compressor based on the fixed step delta-compressor proposed by Engelson et al. [7]. They used this compressor to compress the 64-bit floating-point output of an ordinary differential equation solver. Beside ours, it is the only special-purpose compressor we are aware of that is designed specifically for IEEE double-precision data. We chose a difference order of four for the fsd compressor, which yields the best results on our data sets. Higher difference orders make the (de)compressor slower and degrade the compression ratio.

rar [22] is an increasingly popular compressor that incorporates a combination of Huffman [24], LZ77 [35], and Prediction by Partial Matching [32] algorithms. We use the rar 3.51 executable for Intel-Linux platforms.

7-zip [16] is another compressor that has gained wide acceptance in the community. It is based on the Lempel-Ziv-Markov Chain Algorithm (LZMA) [15], which uses a dictionary compression scheme similar to LZ77. p7zip [13] is a version of 7-zip that is specifically designed for Intel-Linux platforms. We use the 7za stand-alone executable of p7zip in this study.

lzpx [14] is a relatively small compressor written in C++. Its source code size is comparable to that of our compressor, which is the smallest compressor that we investigated (about 300 lines of C code). lzpx is based on the Lempel-Ziv compression algorithm and uses arithmetic coding [30].

zzip [12] is a compressor based on the Burrows-Wheeler Transform (BWT) [2] and is written in C. Note that zzip does not work correctly on one of our data sets (sppm).

gzip [18] and bzip2 [17] are widely-used UNIX file compression tools. gzip's main algorithm is DEFLATE [19], which is a combination of LZ77 and Huffman coding, whereas bzip2 is based on Burrow-Wheeler's block-sorting algorithm that groups bytes with similar contexts and compresses them with a Huffman coder.

4.4. Data sets

We evaluate the speed and the compression ratio of the algorithms described above on data sets obtained from three NAS Parallel Benchmark (NPB) suite applications [1], three ASCI Purple benchmarks [21], and a modified version of eulag [29]. The NAS Parallel Benchmarks are a set of eight programs derived from computational fluid dynamics applications consisting of five kernels and three pseudo-applications. We use the three pseudo-applications lu, bt, and sp. In addition, we use the three solvers sppm, sweep3d, and aztec from the ASCI Purple suite. eulag is a fluid code developed at the National Center for Atmospheric Research to model a broad range of physical situations. We use a modified version that simulates brain injuries during accidents [6].

Table 1: Information about the data sets

	size (MB)	number of doubles	unique doubles	1st order entropy
aztec	512.1	67,126,652	62,172,126	24.42
bt	254.0	33,298,679	30,928,542	23.67
eulag	135.3	17,730,000	16,832,168	23.97
lu	185.1	24,264,871	24,064,865	24.47
sp	276.7	36,263,232	35,881,610	25.03
sppm	266.1	34,874,483	3,572,226	11.24
sweep3d	119.9	15,716,403	14,113,187	23.41

The NPB and ASCI benchmarks are parallel programs for which we recorded the numeric messages sent by (the randomly selected) Node 17. eulag is a sequential program from which we recorded the simulation results after each time step. Table 1 provides relevant information about the resulting data sets. The seven data sets are between about 100 and 500 megabytes in length, which is sufficient to warm up the compressor models and to obtain accurate timing information. With the exception of sppm, the values in each data set are largely unique, i.e., most values occur only once.

5. Results

5.1. Compression and decompression speed

This section evaluates how fast the seven compressors from Section 4.3 and our own (dfcm) can compress and decompress the seven data sets on our reference machine. Since this paper focuses on fast compression and decompression, we use the fastest version of each algorithm in all cases where the speed can be selected in the command line (e.g., with the "--fast" option). Figure 3 shows the geometric mean runtimes for each algorithm in seconds (lower numbers are better). Table 2 lists the compression throughput in number of doubles processed per second (higher numbers are better) for each algorithm and data set. Table 3 is similar to Table 2 except it shows the decompression throughput. The numbers in bold print mark the best results in both tables.

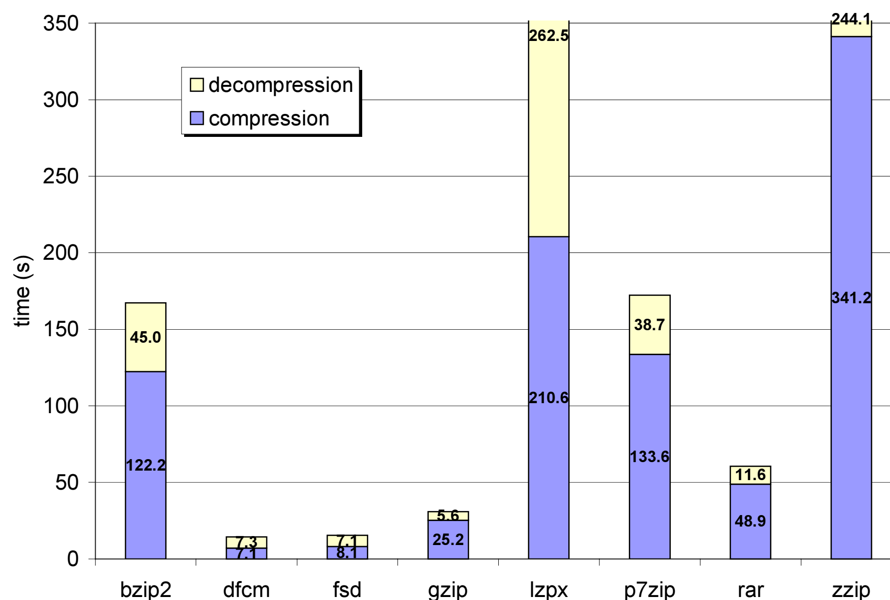

Figure 3: Geometric-mean compression and decompression time in seconds

Figure 3 illustrates that our algorithm is the fastest compressor/decompressor on average. fsd, the other special-purpose compressor, is nearly as fast (6% slower). The fastest general-purpose compressor (gzip) is over twice slower.

Looking at the compression time only, we find dfcm to be 15% faster than fsd and 257% faster than gzip while at the same time also yielding higher average compression ratios (Section 5.2). However, dfcm is 2.5% slower than fsd and 31% slower than gzip at decompression. All other algorithms we studied are slower at compression and decompression than our algorithm.

Looking at the throughput results in Table 2 and Table 3, we find that dfcm compresses our data sets at a rate of over 4.1 million doubles per second on average, compared to about 3.6 million for fsd and just over one million for gzip.

Due to the symmetry of our algorithm, its decompression throughput is almost the same as its compression throughput. Overall, gzip and fsd are the only algorithms that achieve a higher decompression throughput than dfcm. On sppm, which is the only data

set that does not exhibit high first-order entropy and contains relatively few unique values (Table 1), rar, fsd and gzip outperform dfcm.

Table 2: Compression throughput in doubles per second

	bzip2	dfcm	fsd	gzip	lzpx	p7zip	rar	zzip
aztec	309,653.3	**4,853,698.6**	4,068,281.9	851,861.1	89,317.6	204,873.0	407,198.4	90,363.7
bt	201,859.1	**3,899,142.7**	3,394,360.8	798,721.0	82,351.1	207,597.7	699,258.3	83,482.5
eulag	193,791.7	2,832,268.4	**3,390,057.4**	755,754.5	77,872.5	155,240.3	330,413.7	87,223.9
lu	197,435.9	**4,024,025.0**	3,370,121.0	771,292.8	87,274.3	146,544.7	688,365.1	83,579.7
sp	196,017.5	**4,291,506.7**	3,203,465.7	1,201,963.3	123,143.3	200,437.9	419,567.7	67,541.9
sppm	283,417.2	**5,951,277.0**	4,637,564.2	5,046,958.5	983,765.4	646,063.0	2,478,641.3	---
sweep3d	352,781.2	**3,988,934.8**	3,424,052.9	1,237,512.0	172,046.0	200,925.6	423,736.9	93,572.3
geo_mean	240,745.9	**4,171,319.1**	3,612,282.6	1,168,666.5	139,766.6	220,290.6	601,655.0	83,850.9

Table 3: Decompression throughput in doubles per second

	bzip2	dfcm	fsd	gzip	lzpx	p7zip	rar	zzip
aztec	608,748.1	**4,750,647.7**	4,661,573.1	3,773,280.0	68,249.4	592,520.5	2,011,586.8	114,277.6
bt	499,305.4	4,085,727.5	3,858,479.6	**4,157,138.5**	67,270.1	606,202.1	2,021,777.7	106,375.4
eulag	466,578.9	2,850,482.3	**4,038,724.4**	4,011,312.2	62,214.9	503,407.2	1,870,253.2	103,436.2
lu	466,991.4	4,249,539.6	**4,372,048.8**	4,057,670.7	68,463.6	529,222.9	2,962,743.7	100,425.8
sp	484,673.0	4,416,958.8	3,527,551.8	**6,115,216.2**	109,871.9	677,690.7	1,869,238.8	113,033.0
sppm	2,245,620.3	4,381,216.5	4,719,145.2	**11,509,730.4**	755,185.9	3,001,246.4	7,404,348.8	---
sweep3d	710,185.4	3,909,553.0	3,988,934.8	**6,467,655.6**	137,213.2	753,784.3	2,115,262.9	181,587.6
geo_mean	654,032.4	4,047,259.8	4,146,909.1	**5,291,933.5**	112,113.7	760,191.0	2,527,952.7	117,202.0

5.2. Compression ratio

Table 4 lists the compression ratio that the eight compressors achieve on the seven data sets (higher numbers are better). The numbers in bold print mark the highest ratios.

Table 4: Compression ratio

	bzip2	dfcm	fsd	gzip	lzpx	p7zip	rar	zzip
aztec	1.15	**1.69**	1.42	1.22	1.15	1.39	1.26	1.15
bt	1.10	**1.36**	1.02	1.13	1.10	1.32	1.15	1.10
eulag	1.04	**1.23**	1.06	1.06	1.05	1.15	1.07	1.09
lu	1.02	**1.23**	0.99	1.05	1.03	1.22	1.07	1.03
sp	1.08	1.25	0.95	1.11	1.07	**1.31**	1.14	1.07
sppm	6.78	4.16	2.14	6.31	7.94	**8.31**	7.68	---
sweep3d	1.06	**1.49**	1.20	1.09	1.19	1.26	1.30	1.35
geo_mean	1.40	1.60	1.20	1.42	1.46	**1.66**	1.52	1.13

With the exception of sppm, which has low first-order entropy, none of the data sets can be compressed by more than a factor of 1.7 with any of the eight algorithms. It appears that the floating-point results produced by numeric programs tend to be difficult to compress effectively, especially when speed matters.

Our algorithm achieves the highest compression ratio on five of the seven data sets and is a close second behind p7zip on sp. This is a nice result because our algorithm is also one of the fastest. Note that all algorithms except fsd outperform dfcm on sppm, indicating that our algorithm is probably only preferable in situations where high-entropy data is expected.

5.3. Memory usage

Table 5 shows how much memory the eight algorithms maximally allocate when compressing and decompressing the seven data sets. All algorithms use a reasonable amount of memory. No algorithm requires more than about 32 megabytes, which is small for a computer used to perform numeric calculations. dfcm allocates close to 17 megabytes with 16 megabytes in the hash table. However, due to the infrequency with which most of the hash table entries are accessed, the active working set of our algorithm is much smaller and appears not to result in slow processing due to poor cache performance.

Table 5: Memory usage in kilobytes

	bzip2	dfcm	fsd	gzip	lzpx	p7zip	rar	zzip
compression	1,520	16,748	420	636	18,204	2,756	32,252	6,832
decompression	856	16,736	424	444	18,204	2,256	5,476	5,747

6. Summary and conclusion

In situations where large amounts of scientific floating-point data need to be compressed, transferred, and decompressed in a speedy manner, our dfcm algorithm performs very well. It is one of the fastest algorithms and often results in the highest compression ratio. It compresses and decompresses IEEE double-precision values on average at rates of over 32 megabytes per second on a 3GHz Pentium 4. This rate exceeds the throughput of Fast Ethernet networks and of many hard disks, meaning that our algorithm makes real-time compression and decompression possible. In fact, the time saved due to sending or storing the shorter compressed data stream is likely to outweigh the runtime overhead introduced by performing the compression. Hence, our algorithm may well compress the data and reduce the overall runtime simultaneously.

7. Acknowledgment

This work was supported by the National Science Foundation under Grant No. 0125987.

References

[1] D. Bailey, T. Harris, W. Saphir, R. v. d. Wijngaart, A. Woo and M. Yarrow, *The NAS Parallel Benchmarks 2.0*, *Technical Report NAS-95-020*, NASA Ames Research Center, 1995.

[2] M. Burrows and D. J. Wheeler, *A Block-Sorting Lossless Data Compression Algorithm*, *Digital SRC Research Report 124*, 1994.

[3] M. Burtscher, *VPC3: A Fast and Effective Trace-Compression Algorithm*, *Joint International Conference on Measurement and Modeling of Computer Systems*, 2004, pp. 167-176.

[4] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan and N. B. Sam, *The VPC Trace-Compression Algorithms, IEEE Transactions on Computers*, Vol. 54, 2005, pp. 1329-1344.

[5] M. Burtscher and M. Jeeradit, *Compressing Extended Program Traces Using Value Predictors*, *12th International Conference on Parallel Architectures and Compilation Techniques*, 2003, pp. 159-169.

[6] M. Burtscher and I. Szczyrba, *Numerical Modeling of Brain Dynamics in Traumatic Situations - Impulsive Translations*, *The 2005 International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences*, 2005, pp. 205-211.

[7] V. Engelson, D. Fritzson and P. Fritzson, *Lossless Compression of High-Volume Numerical Data from Simulations*, *Data Compression Conference*, 2000, pp. 574-586.
[8] M. Forum, *MPI: A Message-Passing Interface Standard, The International Journal of Supercomputer Applications and High Performance Computing*, 1994, pp. 165-414.
[9] M. N. Gamito and M. S. Dias, *Lossless Coding of Floating Point Data with JPEG 2000 Part 10*, *Applications of Digital Image Processing XXVII*, 2004, pp. 276-287.
[10] F. Ghido, *An Efficient Algorithm for Lossless Compression of IEEE Float Audio*, *Data Compression Conference*, 2004, pp. 429-438.
[11] B. Goeman, H. Vandierendonck and K. Bosschere, *Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency*, *Seventh International Symposium on High Performance Computer Architecture*, 2001, pp. 207-216.
[12] http://debin.net/zzip/, 2002.
[13] http://p7zip.sourceforge.net/, 2005.
[14] http://sourceforge.net/projects/lzpx/, 2005.
[15] http://tukaani.org/lzma/, 2005.
[16] http://www.7-zip.org/, 2005.
[17] http://www.bzip.org/, 2005.
[18] http://www.gzip.org/, 2005.
[19] http://www.ietf.org/rfc/rfc1951.txt, 1996.
[20] http://www.intel.com/design/pentium/MANUALS/24319101.PDF, 2005.
[21] http://www.llnl.gov/asci/platforms/purple/, 2005.
[22] http://www.rarsoft.com/, 2005.
[23] http://www.tc.cornell.edu/, 2005.
[24] D. A. Huffman, *A Method for the Construction of Minimum Redundancy Codes*, Proceedings of the IERE, Vol. 40, 1952, pp. 1098-1101.
[25] M. Isenburg, P. Lindstrom and J. Snoeyink, *Lossless Compression of Floating-Point Geometry*, *CAD2004*, 2004, pp. 495-502.
[26] J. Ke, M. Burtscher and E. Speight, *Runtime Compression of MPI Messages to Improve the Performance and Scalability of Parallel Applications*, *High-Performance Computing, Networking and Storage Conference*, 2004, pp. 59-65.
[27] S. Klimenko, B. Mours, P. Shawhan and A. Sazonov, *Data Compression Study with the E2 Data*, *LIGO-T010033-00-E Technical Report*, 2001, pp. 1-14.
[28] M. H. Lipasti, C. B. Wilkerson and J. P. Shen, *Value Locality and Load Value Prediction*, *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 138-147.
[29] J. M. Prusa, P. K. Smolarkiewicz and A. A. Wyszogrodzki, *Simulations of Gravity Wave Induced Turbulence Using 512 PE CRAY T3E, International Journal of Applied Mathematics and Computational Science*, Vol. 11, 2001, pp. 101-115.
[30] J. Rissanen and G. G. Langdon Jr., *Arithmetic Coding, IBM Journal of Research and Development*, Vol. 23, 1979, pp. 149-162.
[31] Y. Sazeides and J. E. Smith, *The Predictability of Data Values*, *30th International Symposium on Microarchitecture*, 1997, pp. 248-258.
[32] D. Shkarin, *PPM: one step to practicality*, *Data Compression Conference*, 2002, pp. 202-211.
[33] A. Trott, R. Moorhead and J. McGenley, *Wavelets Applied to Lossless Compression and Progressive Transmission of Floating Point Data in 3-D Curvilinear Grids*, *IEEE Visualization*, 1996, pp. 355-388.
[34] B. E. Usevitch, *JPEG2000 Extensions for Bit Plane Coding of Floating Point Data*, *Data Compression Conference*, 2003, pp. 451-461.
[35] J. Ziv and A. Lempel, *A Universal Algorithm for Data Compression, IEEE Transactions on Information Theory*, Vol. 23, 1977, pp. 337-343.