High Throughput Compression of Double-Precision Floating-Point Data

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

Abstract

This paper describes FPC, a lossless compression algorithm for linear streams of 64-bit floating-point data. FPC is designed to compress well while at the same time meeting the high throughput demands of scientific computing environments. On our thirteen datasets, it achieves a substantially higher average compression ratio than BZIP2, DFCM, FSD, GZIP, and PLMI. At comparable compression ratios, it compresses and decompresses 8 to 300 times faster than the other five algorithms.

1. Introduction

Many scientific programs produce and transfer large amounts of double-precision floating-point data. For example, some exchange data between processing nodes and with mass storage devices after every simulated time step. Moreover, scientific programs are usually checkpointed at regular intervals so that they can be restarted from the most recent checkpoint after a crash. Checkpoints tend to be large and have to be saved to disk.

Compression can reduce the amount of data that needs to be transferred and stored. If done fast enough, it can also increase the throughput of the data exchanges, which is desirable in high-performance computing environments. The challenge is to achieve a good compression ratio and a high compression and decompression speed at the same time. Furthermore, the compression algorithm usually has to be lossless and single pass. For example, checkpoints cannot be lossy and neither can data from which certain derived quantities will be computed [16]. To avoid first writing the uncompressed data to disk, a single-pass algorithm is needed so that the data can be compressed and decompressed as it is generated and consumed, respectively.

This paper presents FPC, a lossless, single-pass, linear-time compression algorithm for double-precision floating-point data. FPC is specifically designed for scientific and high-performance computing environments. It delivers a good average compression ratio on hard-to-compress 1D numeric data. Moreover, it employs a simple algorithm that can be implemented entirely with fast integer operations. As a result, FPC compresses and decompresses one to two orders of magnitude faster than other algorithms.

The rest of this paper is organized as follows. Section 2 explains the FPC algorithm in detail. Section 3 summarizes related work. Section 4 discusses the evaluation methods. Section 5 presents the results. Section 6 concludes the paper with a summary.

2. The FPC Algorithm

FPC compresses linear sequences of IEEE 754 double-precision floating-point values by repeatedly predicting the next double in the sequence, xoring the double with the predicted value, and leading-zero compressing the result. As illustrated in Figure 1, it uses

an *fcm* [19] and a *dfcm* [8] value predictor to predict the doubles, both of which are essentially hash tables. The closer of the two predictions, i.e., the one that shares more common most significant bits with the true double, is chosen and xored with the double. The xor operation turns identical bits into zeros. Hence, if the prediction is accurate, the xor result has many leading zero bits. FPC then counts the number of leading zero bytes and encodes the count in a three-bit value along with a one-bit value that specifies which predictor was used. The resulting four-bit code and the nonzero remainder bytes are written to the compressed stream. The latter are emitted verbatim without any form of encoding.

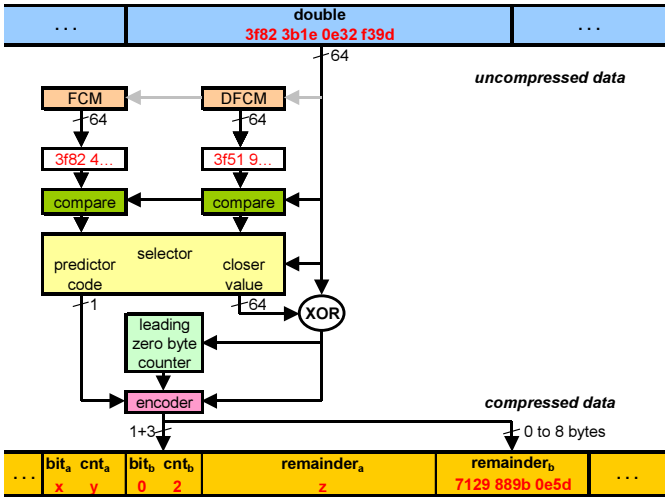

Figure 1: The FPC compression algorithm

To maintain byte granularity, which is much more efficient than bit granularity, a pair of doubles is always processed together and the corresponding two four-bit codes are packed into a byte. In case an odd number of doubles needs to be compressed, a spurious double is encoded at the end. The extra value is expressed as seven leading zero bytes and the "nonzero" byte is set to zero. This encoding is never used with an actual double as it would simply be encoded as having eight leading zero bytes.

Decompression works as follows. It starts by reading the next four-bit code. Then the number of remainder bytes specified by the three-bit value are read and zero-extended to a full 64-bit number. Based on the one-bit value, this number is xored with the 64-bit *fcm* or *dfcm* prediction to recreate the original double. This lossless reconstruction is possible because xor is a reversible operation.

For performance reasons, FPC interprets all doubles as 64-bit integers and uses only integer arithmetic. Since there can be between zero and eight leading zero bytes, i.e., nine possibilities, not all of them can be encoded in the three-bit value. We decided not to support a leading zero count of four because it occurs only rarely. Consequently, all xor results with four leading zero bytes are expressed as having only three leading zero bytes and the fourth zero byte is emitted as part of the remainder bytes.

Before compression and decompression, both predictors are initialized with all zeros. After each prediction, they are updated with the true double value to ensure that they generate the same sequence of predictions during compression as they do during decompression. The following pseudo code demonstrates the operation of the *fcm* predictor. The `table_size` has to be a power of two and `fcm` is the hash table.

```
unsigned long long true_value, fcm_prediction, fcm_hash, fcm[table_size];
...
fcm_prediction = fcm[fcm_hash];  // prediction: read hash table entry
fcm[fcm_hash] = true_value;      // update: write hash table entry
fcm_hash = ((fcm_hash << 6) ^ (true_value >> 48)) & (table_size - 1);
```

Right shifting the `true_value`, i.e., the current double expressed as a 64-bit integer, by 48 bits eliminates the usually random mantissa bits. The remaining 16 bits are xored with the previous hash value to produce the new hash. However, the previous hash is first shifted by six bits to the left to gradually phase out bits from older values. The hash value can therefore be thought of as representing the most recently encountered doubles, and the hash table stores the double that follows this sequence. Hence, making an *fcm* prediction is tantamount to performing a table lookup to determine which value followed the last time a similar sequence of previous doubles was seen.

The *dfcm* predictor operates in much the same way. However, it predicts integer differences between consecutive values rather than absolute values, and the shift amounts in the hash function are different.

```
unsigned long long last_value, dfcm_prediction, dfcm_hash, dfcm[table_size];
...
dfcm_prediction = dfcm[dfcm_hash] + last_value;
dfcm[dfcm_hash] = true_value - last_value;
dfcm_hash = ((dfcm_hash << 2) ^ ((true_value - last_value) >> 40)) &
  (table_size - 1);
last_value = true_value;
```

The complete C source code and a brief description of how to compile and use it are available at http://www.csl.cornell.edu/~burtscher/research/FPC/.

3. Related Work

Our work concentrates on 64-bit floating-point values, such as those produced by numeric programs, which are also the target of the following algorithms from the literature.

Engelson et al. [5] propose a compression scheme for the double-precision output of a numerical solver for ordinary differential equations. The authors use integer delta and extrapolation algorithms to compress and decompress the data. Their method is particularly beneficial with gradually changing data.

Lindstrom and Isenburg [16] designed a scheme for the efficient compression of imaging data, with an emphasis on 2D and 3D data. They predict the data using the Lorenzo predictor [12] and encode the residual, i.e., the difference between the predicted and the true value, with a range coder based on Schindler's quasi-static probability model [20].

Together with Jian Ke, we have previously proposed the DFCM compressor [18], which performs data prediction, residual generation, and backend coding. The data prediction utilizes a modified *dfcm* value predictor. A four-bit leading zero suppress scheme is then employed to encode the residual, which is the xored difference between the true and the predicted value. Another paper [14] describes how we incorporated the DFCM compression algorithm into an MPI library to speed up parallel message passing programs running on a cluster of workstations.

Several papers on lossless compression of floating-point data focus on 32-bit single-precision values, as exemplified by the following work. Klimenko et al. [15] present a method that combines differentiation and zero suppression to compress floating-point

data arising from experiments conducted at the Laser Interferometer Gravitation Wave Observatory. 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. Ghido [7] proposes an algorithm for the lossless compression of audio data. It transforms the floating-point values into integers and generates an additional binary stream for the lossless reconstruction of the original floating-point values.

Several publications concentrate on compressing floating-point data that represent images. These studies focus on maximizing the compression ratio, as the compression and decompression speed are not so important. Usevitch [22] proposes extensions to the JPEG2000 standard that allow data to be efficiently encoded with bit-plane coding algorithms where the floating-point values are represented as "big integers". Gamito et al. [6] describe modifications needed in JPEG2000 to accommodate lossless floating-point compression, namely, adjustments in the wavelet transformation and earlier signaling of special numbers such as NaNs in the main header. Isenburg et al. [13] 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. [21] use an extended precision algorithm, the Haar wavelet transform, and Huffman coding to losslessly compress 3D curvilinear grids. Chen et al. [4] compress irregular grid volume data represented as a tetrahedral mesh. Their technique performs differential coding and clustering to generate separate data residuals for the mantissa and the exponent. Then, a Huffman coder and GZIP are used to encode the mantissa and exponent residuals.

4. Evaluation Methodology

4.1 System and Compiler

We compiled and evaluated FPC and the compressors listed in Section 4.4 on a 64-bit system with a 1.6GHz Itanium 2 CPU, which has a 16kB L1 data cache, a 256kB unified L2 cache, a 3MB L3 cache (on chip), and 3GB of main memory. The operating system is Red Hat Enterprise Linux AS4 and the compiler is the Intel C Itanium Compiler version 9.1. We used the "-O3 -mcpu=itanium2 -static" compiler flags for each compressor.

4.2 Timing Measurements

All timing measurements refer to the elapsed time reported by the shell command *time*. To make the measurements independent of the disk speed, each experiment was conducted five times in a row and the shortest running time is reported. This approach resulted in close to 100% CPU utilization because the compressors' inputs were cached in main memory. All output was written to /dev/null, that is, it was consumed but ignored.

4.3 Datasets

We used thirteen datasets from various scientific domains for our evaluation. Each dataset consists of a one-dimensional binary sequence of IEEE 754 double-precision floating-point numbers and belongs to one of the following categories.

Observational data: These datasets comprise measurements from scientific instruments.

- *obs_error*: data values specifying brightness temperature errors of a weather satellite
- *obs_info*: latitude and longitude of the observation points of a weather satellite
- *obs_spitzer*: data from the Spitzer Space Telescope showing a slight darkening as an extrasolar planet disappears behinds its star

- *obs_temp*: data from a weather satellite denoting how much the observed temperature differs from the actual contiguous analysis temperature field

Numeric simulations: These datasets are the results of numeric simulations.

- *num_brain*: simulation of the velocity field of a human brain during a head impact
- *num_comet*: simulation of the comet Shoemaker-Levy 9 entering Jupiter's atmosphere
- *num_control*: control vector output between two minimization steps in weather-satellite data assimilation
- *num_plasma*: simulated plasma temperature of a wire array z-pinch experiment

Parallel messages: These datasets capture the messages sent by a node in a parallel system running NAS Parallel Benchmark (NPB) [1] and ASCI Purple [11] applications.

- *msg_bt*: NPB computational fluid dynamics pseudo-application bt
- *msg_lu*: NPB computational fluid dynamics pseudo-application lu
- *msg_sp*: NPB computational fluid dynamics pseudo-application sp
- *msg_sppm*: ASCI Purple solver sppm
- *msg_sweep3d*: ASCI Purple solver sweep3d

Table 1 summarizes information about each dataset. The first two data columns list the size in megabytes and in millions of double-precision values. The middle column shows the percentage of doubles in each dataset that are unique, i.e., appear exactly once. The fourth column displays the first-order entropy of the doubles in bits. The last column expresses the randomness of the datasets in percent, that is, it reflects how close the first-order entropy is to that of a truly random dataset with the same number of doubles.

Table 1: Statistical information about each dataset

	size (megabytes)	doubles (millions)	unique values (percent)	1st order entropy (bits)	randomness (percent)
msg_bt	254.0	33.30	92.9	23.67	94.7
msg_lu	185.1	24.26	99.2	24.47	99.7
msg_sp	276.7	36.26	98.9	25.03	99.7
msg_sppm	266.1	34.87	10.2	11.24	44.9
msg_sweep3d	119.9	15.72	89.8	23.41	97.9
num_brain	135.3	17.73	94.9	23.97	99.5
num_comet	102.4	13.42	88.9	22.04	93.1
num_control	152.1	19.94	98.5	24.14	99.6
num_plasma	33.5	4.39	0.3	13.65	61.9
obs_error	59.3	7.77	18.0	17.80	77.8
obs_info	18.1	2.37	23.9	18.07	85.3
obs_spitzer	189.0	24.77	5.7	17.36	70.7
obs_temp	38.1	4.99	100.0	22.25	100.0

We observe that all datasets contain several million doubles. What is striking is that the datasets from all three categories appear to largely consist of unique values. Moreover, they are highly random from an entropy perspective, even the ones that do not contain many unique values (e.g., *num_plasma*).

Based on these statistics, it is unlikely that a pure entropy-based compression approach would work well. Note that the higher-order entropies are also close to random because of the large percentage of unique values. Clearly, we have to use a good data model or subdivide the doubles into smaller entities (e.g., bytes), some of which may exhibit less randomness, to compress these datasets well. FPC incorporates both of these approaches.

4.4 Other Compressors

This subsection describes the compression schemes with which we compare our approach in Section 5. GZIP and BZIP2 are lossless, general-purpose algorithms that can be used to compress any kind of data. The remaining algorithms represent our implementations of special-purpose floating-point compressors from the literature. They are all single-pass, lossless compression schemes that "know" about the format of double-precision values.

BZIP2: BZIP2 [9] is a general-purpose compressor that operates at byte granularity. It implements a variant of the block-sorting algorithm described by Burrows and Wheeler [2]. It applies a reversible transformation to a block of inputs, uses sorting to group bytes with similar contexts together, and then compresses them with a Huffman coder. The block size is adjustable. We evaluate all supported block sizes, i.e., one through nine.

DFCM: Our previously proposed DFCM scheme [18] maps each encountered floating-point value to an unsigned integer and predicts it with a modified *dfcm* predictor. This predictor computes a hash value out of the three most recently encountered differences between consecutive values in the input. Next, it performs a hash table lookup to retrieve the differences that followed the last two times the same hash was encountered, and one of the two differences is used to predict the next value. A residual is generated by xoring the predicted value with the true value. This residual is encoded using a four-bit leading zero bit count. We evaluate predictor sizes between 16 bytes and 512MB. Note that DFCM and FPC utilize quite different *dfcm* predictor implementations.

FSD: The FSD compressor implements the fixed step delta-algorithm proposed by Engelson et al. [5]. As it reads in a stream of doubles, it iteratively generates difference sequences from the original sequence. The order determines the number of iterations. A zero suppress algorithm is then used to encode the final difference sequence, where each value is expected to have many leading zeroes. Generally, gradually changing data tend to benefit from higher difference orders whereas rapidly changing data compress better with lower orders. We evaluate orders one through seven (higher orders perform worse).

GZIP: GZIP [10] is a general-purpose compression utility that operates at byte granularity and implements a variant of the LZ77 algorithm [23]. It looks for repeating sequences of bytes (strings) within a 32kB sliding window. The length of the string is limited to 256 bytes, which corresponds to the lookahead buffer size. GZIP uses two Huffman trees, one to compress the distances in the sliding window and another to compress the lengths of the strings as well as the individual bytes that were not part of any matched sequence. The algorithm finds duplicated strings using a chained hash table. A command-line argument determines the maximum length of the hash chains and whether lazy evaluation should be used. We evaluate all supported levels, i.e., one through nine.

PLMI: The PLMI scheme proposed by Lindstrom and Isenberg [16] uses a Lorenzo predictor in the front-end to predict 2D and 3D geometry data for rendering. Since our datasets are 1D, we cannot evaluate PLMI in its intended mode. In fact, for general linear data, the Lorenzo predictor reverts to a delta predictor, which processes data similarly to the first-order FSD algorithm. Hence, we included the modified *dfcm* predictor (see above) in our implementation of PLMI, which compresses linear data better. The predicted and true floating-point values are mapped to an unsigned integer from which a residual is computed by a difference process. The final step involves encoding the residual with range coding based on Schindler's quasi-static probability model. We evaluate predictor sizes between 16 bytes and 512MB.

5. Results

5.1 Compression Ratio

This subsection investigates the highest compression ratio that the six algorithms achieve on each dataset. Note that we individually optimized the size (DFCM, FPC, PLMI), level (BZIP2, GZIP), or order (FSD) for each algorithm and dataset to obtain the results shown in Table 2. The numbers in bold print reflect the best compression ratio for each dataset.

Table 2: Highest compression ratio of the six algorithms on each dataset

	BZIP2	DFCM	FPC	FSD	GZIP	PLMI
msg_bt	1.10	**1.36**	1.29	1.07	1.13	1.24
msg_lu	1.02	**1.24**	1.17	1.00	1.06	1.19
msg_sp	1.08	1.25	**1.26**	0.99	1.11	1.19
msg_sppm	6.93	4.23	5.30	2.35	**7.43**	5.02
msg_sweep3d	1.29	1.56	**3.09**	1.21	1.09	1.21
num_brain	1.04	**1.23**	1.16	1.10	1.06	1.12
num_comet	1.17	1.17	1.16	1.11	1.16	**1.18**
num_control	1.03	**1.07**	1.05	0.99	1.06	1.06
num_plasma	5.79	1.30	**15.05**	1.00	1.61	1.26
obs_error	1.34	1.52	**3.60**	1.16	1.45	1.26
obs_info	1.22	1.23	**2.27**	1.00	1.15	1.16
obs_spitzer	**1.75**	1.00	1.03	0.96	1.23	1.08
obs_temp	1.02	1.01	1.02	0.97	**1.04**	1.04
geo_mean	*1.52*	*1.36*	***1.95***	*1.11*	*1.35*	*1.30*

FPC achieves the highest geometric-mean compression ratio because on four datasets it exceeds the performance of the other five algorithms by a large margin. The other algorithms substantially outperform FPC on two datasets, *msg_sppm* and *obs_spitzer*.

DFCM is sometimes superior to FPC because it employs a more sophisticated predictor, which stores two difference values in each table entry (instead of just a single value as FPC's predictors do) and uses a more elaborate hash function. However, FPC outperforms DFCM on the majority of our datasets because FPC contains a second predictor that often complements the first predictor well (e.g., on *num_plasma*).

No algorithm performs best on all datasets. In fact, no algorithm is best on more than five of the thirteen datasets. There is also no best algorithm within the three dataset categories. Even GZIP and BZIP2, the general-purpose compressors that have no knowledge of the format of double-precision floating-point values, provide the highest compression ratio on some of the datasets. Only FSD is outperformed on all of our datasets.

None of our datasets are highly compressible with any of the algorithms we studied. Only *msg_sppm* can be compressed by at least a factor of two with all six algorithms. Two datasets, *num_control* and *obs_temp*, cannot even be compressed by ten percent. These results are consistent with the randomness information presented in Table 1, based on which we would expect *msg_sppm* to be the most and *obs_temp* the least compressible dataset. The highest overall compression ratio of 15.05 is obtained on *num_plasma*, which exhibits the second lowest randomness and the lowest percentage of unique values.

On some datasets, most notably *msg_sweep3d*, *num_plasma*, *obs_error*, and *obs_info*, and to a lesser extent *obs_spitzer*, one algorithm performs much better than the others. With the exception of *msg_sweep3d*, these datasets all have relatively few unique values and low randomness. The five datasets with above 99% randomness cannot be compressed by more than 26% by any of the algorithms we investigated.

5.2 Throughput

This subsection examines the compression and decompression throughput of the six algorithms (i.e., the raw dataset size divided by the runtime). Figure 2 plots the throughput in gigabits per second versus the geometric-mean compression ratio. For DFCM, FPC, and PLMI, the table size doubles for each data point from sixteen bytes (leftmost) to 512MB (rightmost). For BZIP2 and GZIP, the data points correspond to levels one (leftmost) through nine (rightmost). For FSD, the figure shows results for order one (rightmost) through order seven (leftmost). All other parameters are fixed.

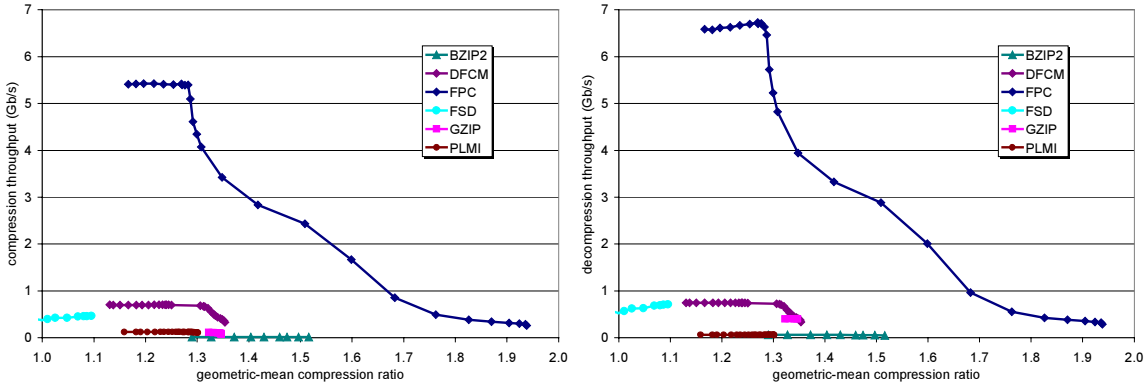

Figure 2: Average compression (left) and decompression (right) throughput versus the geometric-mean compression ratio over the thirteen datasets

For a given compression ratio, FPC compresses our datasets 8 to 300 times faster and decompresses them 9 to 100 times faster than the other algorithms. DFCM has the second highest throughput though GZIP's decompression throughput is similar. FSD is third, but it delivers the lowest compression ratios on our datasets. PLMI compresses the datasets faster than GZIP but decompresses them more slowly. BZIP2 is the slowest algorithm but reaches the second highest compression ratio. All algorithms except our implementation of PLMI decompress faster than they compress. FPC compresses at up to 5.43Gb/s and decompresses at up to 6.73Gb/s.

5.3 Memory Usage

This subsection studies the memory footprint, as reported by the UNIX command *ps*, of the six algorithms. Figure 3 shows the total memory consumption in megabytes relative to the geometric-mean compression ratio. For GZIP and BZIP2, which allocate a different amount of memory for compression and decompression, Figure 3 plots the larger amount. The individual datapoints again correspond to different sizes, levels, or orders.

Except for FPC, all algorithms essentially reach their highest geometric-mean compression ratio with less than ten megabytes. FSD and GZIP have a constant memory footprint. PLMI and DFCM's modified *dfcm* predictor does not benefit from more than six megabytes of memory. At the low end, FPC's memory usage is determined by its code and stack size as well as the input and output buffers. But for larger sizes, the two hash tables dominate, as can be seen from the exponentially growing curve. The same is true for DFCM and PLMI. However, unlike the modified *dfcm* predictor, FPC's two predictors can turn additional memory (up to ten megabytes) into higher compression ratios.

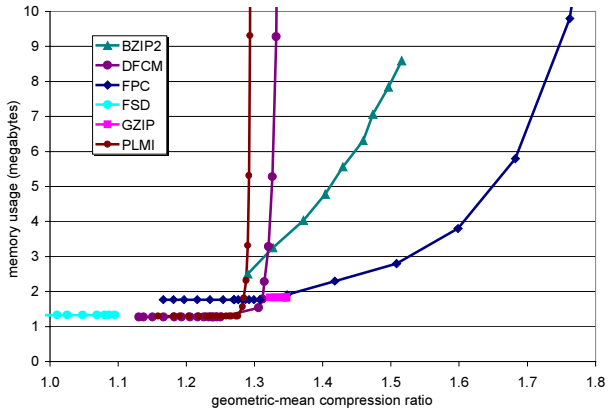

Figure 3: Memory usage versus compression ratio of the six algorithms

6. Summary

This paper describes the lossless FPC compression algorithm for double-precision floating-point data. FPC uses two context based value predictors to predict the next value. The prediction and the true value are xored and the result is leading zero byte compressed. This algorithm was chosen because it is effective and can be implemented efficiently. Varying the predictors' table sizes allows to trade off throughput for compression ratio.

FPC delivers the highest geometric-mean compression ratio and the highest throughput on our thirteen hard-to-compress scientific datasets. It achieves individual compression ratios between 1.02 and 15.05. With tables that fit into the L1 data cache, it delivers a sustained throughput of roughly 100 million doubles per second on a 1.6GHz Itanium 2. This corresponds to only two machine cycles to process a byte of data. The source code is available at http://www.csl.cornell.edu/~burtscher/research/FPC/.

The current version of FPC does not compress multidimensional datasets, 32-bit floating-point values, and easy-to-compress data particularly well. Hence, in future work, we intend to generalize FPC by including an optional second compression stage, providing support for multiple dimensions, and designing a version that is optimized for single-precision data.

7. Acknowledgements

This material is based upon work supported by the Department of Energy under Award Number DE-FG02-06ER25722. Intel Corporation donated the Itanium 2 server. The views and opinions expressed herein do not necessarily state or reflect those of the DOE or Intel. Prof. Joseph Harrington of the Department of Physics at the University of Central Florida provided the datasets *obs_spitzer* and *num_comet*. Prof. David Hammer and Ms. Jiyeon Shin of the Laboratory of Plasma Studies at Cornell University provided *num_plasma*. Mr. Sami Saarinen of the European Centre for Medium-Range Weather Forecasts provided *obs_temp*, *obs_error*, *obs_info*, and *num_control*. *num_brain* was generated using a modified version of EULAG [3], [17], a fluid code developed at the National Center for Atmospheric Research in Boulder, Colorado. Mr. Jian Ke ran the NPB and ASCI Purple benchmarks with 64 processes to capture the message datasets.

8. References

[1] D. Bailey, T. Harris, W. Saphir, R. v. d. Wijngaart, A. Woo and M. Yarrow. "The NAS Parallel Benchmarks 2.0." *Tech. 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*. May 1994.

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

[4] D. Chen, Y.-J. Chiang and N. Memon. "Lossless compression of point-based 3D models." *Pacific Graphics*, pp. 124-126. October 2005.

[5] V. Engelson, D. Fritzson and P. Fritzson. "Lossless Compression of High-Volume Numerical Data from Simulations." *Data Compression Conf.*, pp. 574-586. 2000.

[6] M. N. Gamito and M. S. Dias. "Lossless Coding of Floating Point Data with JPEG 2000 Part 10." *Applications of Digital Image Processing XXVII*, pp. 276-287. 2004.

[7] F. Ghido. "An Efficient Algorithm for Lossless Compression of IEEE Float Audio." *Data Compression Conference*, pp. 429-438. 2004.

[8] B. Goeman, H. Vandierendonck and K. Bosschere. "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency." *International Symposium on High Performance Computer Architecture*, pp. 207-216. 2001.

[9] http://www.bzip.org/, 2006.

[10] http://www.gzip.org/, 2006.

[11] http://www.llnl.gov/asci/purple/benchmarks/limited/code_list.html, 2006.

[12] L. Ibarria, P. Lindstrom, J. Rossignac and A. Szymczak. "Out-of-Core Compression and Decompression of Large n-Dimensional Scalar Fields." *Eurographics*, pp. 343-348. September 2003.

[13] M. Isenburg, P. Lindstrom and J. Snoeyink. "Lossless Compression of Floating-Point Geometry." *CAD2004*, pp. 495-502. 2004.

[14] 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*, pp. 59-65. 2004.

[15] S. Klimenko, B. Mours, P. Shawhan and A. Sazonov. "Data Compression Study with the E2 Data." *LIGO-T010033-00-E Technical Report*, pp. 1-14. 2001.

[16] P. Lindstrom and M. Isenburg. "Fast and Efficient Compression of Floating-Point Data." *IEEE Trans. on Visualization and Computer Graphics*, Vol. 12, No. 5. 2006.

[17] 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, pp. 101-115. 2001.

[18] P. Ratanaworabhan, J. Ke and M. Burtscher. "Fast Lossless Compression of Scientific Floating-Point Data." *Data Compression Conference*, pp. 133-142. 2006.

[19] Y. Sazeides and J. E. Smith. "The Predictability of Data Values." *30th International Symposium on Microarchitecture*, pp. 248-258. December 1997.

[20] M. Schindler. "A Fast Renormalisation for Arithmetic Coding." *Data Compression Conference*, p. 572. March-April 1998.

[21] 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*, pp. 355-388. 1996.

[22] B. E. Usevitch. "JPEG2000 Extensions for Bit Plane Coding of Floating Point Data." *Data Compression Conference*, pp. 451-461. 2003.

[23] J. Ziv and A. Lempel. "A Universal Algorithm for Data Compression." *IEEE Transaction on Information Theory*, Vol. 23, No. 3, pp. 337-343. May 1977.