FPC: A High-Speed Compressor for Double-Precision Floating-Point Data

Martin Burtscher
Center for Grid and Distributed Computing
The University of Texas at Austin
burtscher@ices.utexas.edu

Paruj Ratanaworabhan
Computer Systems Laboratory
Cornell University
paruj@csl.cornell.edu

ABSTRACT

Many scientific programs exchange large quantities of double-precision data between processing nodes and with mass storage devices. Data compression can reduce the number of bytes that need to be transferred and stored. However, data compression is only likely to be employed in high-end computing environments if it does not impede the throughput. This paper describes and evaluates FPC, a fast lossless compression algorithm for linear streams of 64-bit floating-point data. FPC works well on hard-to-compress scientific datasets and meets the throughput demands of high-performance systems. A comparison with five lossless compression schemes, BZIP2, DFCM, FSD, GZIP, and PLMI, on four architectures and thirteen datasets shows that FPC compresses and decompresses one to two orders of magnitude faster than the other algorithms at the same geometric-mean compression ratio. Moreover, FPC provides a guaranteed throughput as long as the prediction tables fit into the L1 data cache. For example, on a 1.6 GHz Itanium 2 server, the throughput is 670 megabytes per second regardless of what data are being compressed.

Index Terms – data compression, prediction methods, data models, floating-point compression

1. INTRODUCTION

Many scientific applications produce and transfer large amounts of 64-bit floating-point data. Some exchange data between processing nodes and with mass storage devices after every simu-

lation time step. In addition, scientific programs are usually checkpointed at regular intervals so that they can be restarted from the most recent checkpoint after a crash. Checkpoint data 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 actually increase the throughput of the data exchanges. Of course, the challenge is to achieve a good compression ratio and a high compression and decompression speed at the same time. Additionally, the compression algorithm should be lossless and single pass. For example, checkpoint data cannot be lossy and neither can data from which certain derived quantities will be computed. A single-pass algorithm is needed so that the data can be compressed and decompressed on the fly as it is generated and consumed, respectively.

This paper presents and evaluates FPC, a lossless, single-pass, linear-time compression algorithm. FPC targets streams of double-precision floating-point data with unknown internal structure, such as the data seen by the network or a storage device in scientific and high-performance computing systems. If the internal structure is known, e.g., a matrix or a linearized tree, then this extra information could be exploited to improve the compression ratio [23]. FPC delivers a good average compression ratio on hard-to-compress numeric data. Moreover, it employs a simple algorithm that is very fast and easy to implement with integer operations. We found FPC to compress and decompress 2 to 300 times faster than the special-purpose floating-point compressors DFCM, FSD and PLMI and the general-purpose compressors BZIP2 and GZIP.

The execution path (i.e., the control flow) through FPC's code is independent of the input data and the compression ratio. Furthermore, in the steady state (i.e., after a short period of compulsory cache misses), all instructions in the algorithm have a fixed latency as long as the prediction tables fit into the L1 data cache. As a consequence, the number of machine cycles needed to compress/decompress a double value is constant, meaning that the time it takes to process a given block of data is known a priori. Hence, FPC provides a constant throughput guarantee when used with small enough table sizes.

The remainder of this paper is organized as follows. Section 2 describes the FPC algorithm and its design in detail. Section 3 summarizes related work. Section 4 explains the evaluation methods. Section 5 presents performance results for FPC and five other compressors. Section 6 concludes the paper with a summary and directions for future work.

2. THE FPC ALGORITHM

2.1 Operation

FPC compresses linear sequences of IEEE 754 double-precision floating-point values by sequentially predicting each value, xoring the true value with the predicted value, and leading-zero compressing the result. As illustrated in Figure 1, it uses variants of an *fcm* [27] and a *dfcm* [14] value predictor to predict the doubles. Both predictors are effectively hash tables. The more accurate of the two predictions, i.e., the one that shares more common most significant bits with the true value, is xored with the true value. The xor operation turns identical bits into zeros. Hence, if the predicted and the true value are close, the xor result has many leading zeros. FPC then counts the number of leading zero bytes, encodes the count in a three-bit value, and concatenates it with a single bit that specifies which of the two predictions was used. The resulting four-bit code and the nonzero residual bytes are written to the output. The latter are emitted verbatim without any encoding.

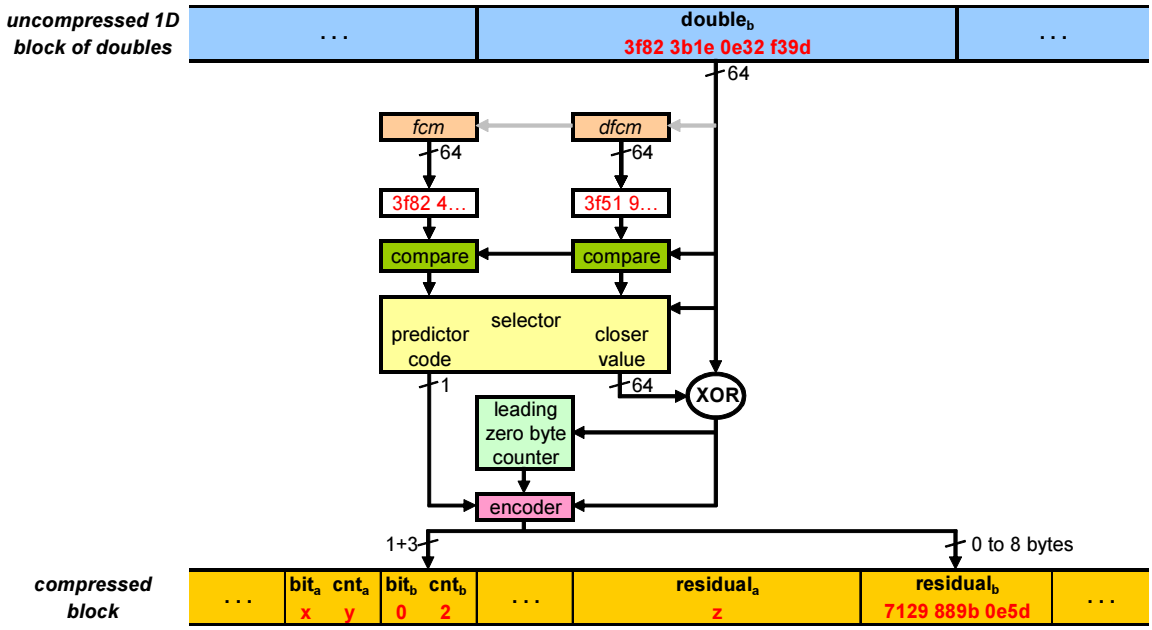


Figure 1: FPC compression algorithm overview

FPC outputs the compressed data in blocks. Each block starts with a header that specifies how many doubles the block encodes and how long it is (in bytes). The header is followed by the stream of four-bit codes, which in turn is followed by the stream of residual bytes. To maintain

byte granularity, which is 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. This spurious value is later eliminated using the count information from the header. Note that our first version of FPC [4] does not use blocks. We added them now because keeping the four-bit codes and the residual bytes separate instead of interleaving them makes FPC faster and potentially simplifies post-processing of the output (e.g., adding another compression stage).

Decompression works as follows. It starts by reading the current four-bit code, decoding the three-bit field, reading the specified number of residual bytes, and zero-extending them to a full 64-bit number. Based on the one-bit field, this number is xored with either the 64-bit *fcm* or *dfcm* prediction to recreate the original double. This lossless reconstruction is possible because xor is reversible.

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 with a three-bit value. We decided not to support a leading zero count of four because it occurs only rarely (cf. Section 5.4). Consequently, all xor results with four leading zero bytes are treated like values with only three leading zero bytes and the fourth zero byte is emitted as part of the residual.

Before compression and decompression, both predictor tables are initialized with 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. `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 `true_value` (i.e., the current double expressed as a 64-bit integer) by 48 bits eliminates the often 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 (`fcm_hash`) therefore represents the sequence of 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 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/. The web site also contains links to our datasets as well as to a detailed discussion of the code and some of the optimization techniques it employs.

2.2 Design

FPC's primary objective is to maximize the throughput while still delivering a competitive compression ratio. Therefore, FPC does not include features that improve the compression ratio at a significant cost of speed. For example, we deemed extracting and handling the sign, exponent and mantissa separately to be too slow for throughput-oriented compression. Likewise, we excluded variable-length encoding at bit granularity as well as bit reversal because of their ineffi-

ciency on modern CPUs. Furthermore, we replaced all floating-point arithmetic with integer arithmetic. Even though the former is more natural and sometimes results in better compression ratios, it is slower and, more importantly, may cause exceptions.

Our previous experience with fast lossless compressors [3], [21], [26] demonstrated algorithms that predict the data using value predictors and leading-zero compress the residual to be very fast while offering a good compression ratio. Hence, we based FPC on this approach. We considered both subtraction and xoring for the residual generation. Since subtraction with a two's complement representation yields about a ten percent lower compression ratio and subtraction with a sign-magnitude representation a three to eight percent lower compression ratio as well as a lower processing speed, we abandoned subtraction and selected xor.

2.2.1 Predictor Parameter Selection

Value predictors have been researched extensively to predict the results of CPU machine instructions at runtime [24]. These predictors are designed to make billions of predictions per second in hardware. As a consequence, they employ simple and fast prediction algorithms.

First, we had to determine which and how many (software) predictors to use. As one might expect, the more accurate prediction algorithms tend to be slower. Similarly, employing a larger number of predictors increases the probability of one of them being correct but lowers the throughput. We experimented with many combinations and configurations of four basic value predictors (a last value [24], a stride [11], a finite context method [27], and a differential finite context method predictor [14]) as well as variations thereof (including a last *n* value [6] and a stride 2-delta predictor [27]).

Because a high processing speed was paramount in our design, we soon found two-predictor combinations to represent the best tradeoff for the following reasons. First, adding predictors increases the runtime linearly but quickly yields diminishing returns on the gained compression ratio. Therefore, only few predictors should be used. Second, to achieve high performance, we had to operate at least at byte granularity. Consequently, we were faced with three-bit codes to express the number of leading zero bytes of the residual between the predicted and the true value. That left five bits to select one of 32 predictors, which was far beyond the number of predictors we could reasonably employ. The only good alternative, which we ended up choosing, was

to utilize one bit to pick between two predictors. Concatenating this bit with the three-bit leading zero count resulted in a four-bit field, which can be combined with the four-bit field of the next prediction to form a byte. (Four-predictor combinations together with two-bit codes for expressing the leading zero counts result in poor compression ratios.)

The next question was which two predictors to select. Initially, we evaluated single predictors with different configurations in isolation and paired up the best performers. However, this approach ended up combining predictors that largely made the same predictions. So we switched to evaluating predictor pairs rather than single predictors, i.e., we optimized the algorithm as a whole instead of its individual components. The result was a significant boost in compression ratio without loss in throughput. Note that the predictors making up the best pairs do not perform particularly well when used in isolation, but they complement each other nicely.

The two-predictor experiments revealed that we should combine an *fcm* predictor with a *dfcm* predictor. That left us with determining good parameters for these predictors. For speed reasons and to prevent overfitting to our datasets, we opted to hardcode the parameters and use the same fixed set of parameters for all predictor sizes. To determine the best configuration, we evaluated the following ten thousand combinations of table sizes and shift amounts in the two hash functions (cf. Section 2.1) on each dataset:

Number of table entries: 1024, 32768, 1048576

Left shift in *fcm* hash function: 1, 2, 3, 4, 5, 6, 7, 8

Right shift in *fcm* hash function: 8, 16, 24, 32, 40, 48, 56

Left shift in *dfcm* hash function: 1, 2, 3, 4, 5, 6, 7, 8

Right shift in *dfcm* hash function: 8, 16, 24, 32, 40, 48, 56

Next, we performed a local search to refine the best right-shift amounts. Unfortunately, no clear winners could be identified because different datasets prefer different configurations and large predictors work well with settings that are suboptimal for small predictors and vice versa. In the end, we settled for the parameters listed in the previous section, which perform reasonable in most cases and work well in the mid range of table sizes.

Because FPC runs at the same speed for all table sizes that fit into the L1 data cache but compresses better with larger tables, there is little reason to use it with very small tables (e.g., less than half of the L1 data cache size). Hence, we were not overly concerned with our parameter

choices resulting in poor compression at the low end. Nevertheless, the most important change to improve the compression ratio with small tables is to increase the right-shift amount in *dfcm* from forty to a value in the fifties. This change would increase the average compression ratio over our datasets by about three percent. At the high end, better hash functions are obtained by lowering the left-shift amount in *fcm* to between two and four, increasing the *dfcm* left-shift amount to between four and eight, and lowering the *dfcm* right-shift amount to 32. This change would increase the average compression ratio by one percent.

3. RELATED WORK

A large body of work related to lossy floating-point compression exists, in particular for the transmission and reproduction of audio and image data. Because some degree of imprecision can be tolerated in these domains, such data do not have to be recreated exactly. Our work focuses exclusively on the lossless compression of floating-point values. There are many instances, especially in science and engineering, where lossless compression is required. Simulation checkpoint data and medical images are examples where lossy compression is unacceptable.

Much of the related work that deals with lossless compression of floating-point data focuses on 32-bit single-precision values. Our work concentrates on 64-bit double-precision data, such as those produced by numeric programs, which are also the target of the following algorithms from the literature.

Engelson et al. [9] propose a compression scheme for the double-precision output of a numerical solver for ordinary differential equations. It uses integer delta and extrapolation techniques to compress and decompress the data. This method is particularly beneficial with gradually changing data. The difference between consecutive values of this nature is small and can, therefore, be encoded with only a few bits. The algorithm includes support for fixed and varying step sizes. It can optionally perform lossy compression.

Lindstrom and Isenburg [23] designed an efficient compressor for both 32- and 64-bit images. Their emphasis is on 2D and 3D data for rendering. The algorithm predicts the data using the Lorenzo predictor [18] and encodes 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 [28].

We have previously used value predictors as data models in program-execution-trace compressors [3]. However, that work focuses on integer data. Together with Jian Ke, the authors have proposed the DFCM compressor for 64-bit floating-point data [26]. DFCM uses a modified *dfcm* value predictor to generate the residual, which is the xored difference between the true and the predicted value. Then, a four-bit leading zero suppress scheme is employed to encode it. We incorporated the DFCM compression algorithm in an MPI library to speed up parallel message-passing programs running on a cluster of workstations [21].

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. [22] present a method that combines differentiation and zero suppression to compress floating-point data from experiments conducted at the Laser Interferometer Gravitation Wave Observatory (LIGO). 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 [13] 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.

Lossless compression of single-precision floating-point data is also of interest to the scientific visualization and imaging community. Several publications cover the compression of the different types of data encountered in this field. These studies, however, focus on maximizing the compression ratio as the compression and decompression speed are not very important.

Fowler et al. [10] use a predictive coding technique to compress volumetric datasets from medical images used for diagnosis and treatment. Their method employs a combination of differential pulse-code modulation (DPCM) and Huffman coding to predict and encode a data sample, respectively.

Ibarria et al. [18] propose the Lorenzo predictor for compressing high-dimensional scalar fields. The predictor is an extension of the two-dimensional parallelogram predictor originally proposed by Touma and Gotsman [29] to compress triangle meshes. The residuals generated by the predictor are further encoded using arithmetic coding. The scheme by Ibarria et al. requires only a small buffer, and is, thus, appropriate for out-of-core compression.

Usevitch [31] 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. [12] 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. [20] describe an adaptation to lossy predictive geometry coding to compress vertex positions in triangular meshes in a lossless manner. The idea is to break up each floating-point value into its sign, exponent, and mantissa component and to compress them separately. This scheme employs the parallelogram predictor proposed by Touma and Gotsman [29] in the prediction stage and a context-based arithmetic coder in the coding stage.

Trott et al. [30] use an extended precision algorithm, the Haar wavelet transform, and Huffman coding to losslessly compress 3D curvilinear grids. The extended precision algorithm first converts single-precision data into double-precision data so that loss of precision will not occur during the transformation and coding process.

Chen et al. [7] 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 Systems and Compilers

We compiled and evaluated FPC and the compressors listed in Section 4.4 on the following four systems.

• A 64-bit **Alpha** system with an 833 MHz Alpha 21264B CPU, a 2-way associative 64 kB L1 data cache, a direct-mapped 4 MB unified L2 cache (off chip), and 1 GB of main memory. The operating system is Tru64 UNIX V5.1B. We used the Compaq C Compiler version 6.5 with the "-O3 -arch ev68 -non_shared" flags.

• A 64-bit **Athlon** system with a 2 GHz Athlon 64 CPU, a 2-way associative 64 kB L1 data cache, a 16-way associative 512 kB unified L2 cache (which does not duplicate the data in the L1 cache), and 1 GB of main memory. The operating system is Red Hat Linux 3.4.5-2 and the

compiler is gcc version 3.4.5. All programs were compiled with the "-O3 -march=athlon64 -static" flags on this system.

• A 64-bit **Itanium** system with a 1.6 GHz Itanium 2 CPU, a 4-way associative 16 kB L1 data cache, an 8-way associative 256 kB unified L2 cache, a 12-way associative 3 MB unified L3 cache (on chip), and 3 GB 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.

• A 32-bit **Pentium** system with a 3 GHz Pentium4-Xeon CPU, a 4-way associative 16 kB L1 data cache, an 8-way associative 1 MB unified L2 cache, and 1 GB of main memory. The operating system is SuSE Linux 9.1 and the compiler is gcc version 3.3.3. We compiled the compressors with the "-O3 -march=pentium4 -static" flags on this machine.

4.2 Timing Measurements

All timing measurements in this paper refer to the elapsed time reported by the UNIX 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. (Using the median runtime instead of the minimum does not change the results significantly.) This approach minimized the timing component due to disk I/O operations because, after the first run, the compressors' inputs were cached in main memory by the operating system. 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 four datasets comprise measurements from scientific instruments.

- *obs_error*: data values specifying brightness temperature errors of a weather satellite
- *obs_info*: latitude and longitude information 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 four datasets are the result 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 evolution of a wire array z-pinch experiment

Parallel messages: These five datasets contain the numeric messages sent by a node in a parallel system running NAS Parallel Benchmark (NPB) [1] and ASCI Purple [17] 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 floating-point values. The middle column shows the percentage of values in each dataset that are unique, i.e., appear exactly once. The fourth column displays the first-order entropy of the values 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 unique values.

The entropy is computed as

$$entropy = -\sum_{i=0}^{n-1}\left(\frac{freq_i}{total}\times\log_2\left(\frac{freq_i}{total}\right)\right)$$

where n is the number of distinct values, *total* refers to the total number of values, and the $freq_i$ are the number of occurrences of each distinct value (i.e., $total = \sum_{i=0}^{n-1} freq_i$). The randomness is

$$randomness = \frac{entropy}{\log_2(n)}$$

where the denominator is the first-order entropy of a sequence of n values that are all distinct.

Table 1: Statistical information about the datasets

	size (megabytes)	doubles (millions)	unique values (percent)	1st order entropy (bits)	randomness (percent)
msg_bt	254.0	33.30	92.9	23.67	95.1
msg_lu	185.1	24.26	99.2	24.47	99.8
msg_sp	276.7	36.26	98.9	25.03	99.7
msg_sppm	266.1	34.87	10.2	11.24	51.6
msg_sweep3d	119.9	15.72	89.8	23.41	98.6
num_brain	135.3	17.73	94.9	23.97	99.9
num_comet	102.4	13.42	88.9	22.04	93.8
num_control	152.1	19.94	98.5	24.14	99.6
num_plasma	33.5	4.39	0.3	13.65	99.4
obs_error	59.3	7.77	18.0	17.80	87.2
obs_info	18.1	2.37	23.9	18.07	94.5
obs_spitzer	189.0	24.77	5.7	17.36	85.0
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 doubles (e.g., *num_plasma*).

Based on these statistics, it is unlikely that a purely entropy-based compression approach will work well. Note that the higher-order entropies (not shown) 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 approaches.

4.4 Compressors

This section describes the compression schemes with which we compare our approach. BZIP2 and GZIP 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. We compiled the C source code of each algorithm described in this section with the same compiler and optimization flags (cf. Section 4.1).

BZIP2: BZIP2 [15] 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]. BZIP2 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 BZIP2 version 1.0.2 with all supported block sizes, i.e., one through nine.

DFCM: Our previously proposed DFCM scheme [26] 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 all predictor sizes between 16 bytes and 512 MB that are powers of two. Note that DFCM and FPC utilize different *dfcm* predictors.

FSD: The FSD compressor implements the fixed step delta-algorithm proposed by Engelson et al. [9]. 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.

GZIP: GZIP [16] is a general-purpose compression utility that operates at byte granularity and implements a variant of the LZ77 algorithm [32]. It looks for repeating strings, i.e., sequences of bytes, within a 32 kB 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 GZIP version 1.3.5 with all supported levels, i.e., one through nine.

PLMI: The PLMI scheme proposed by Lindstrom and Isenburg [23] employs a Lorenzo predictor in the front-end to predict 2D and 3D geometry data for rendering. Since our datasets are one dimensional (i.e., we do not have dimension information), we cannot evaluate PLMI in its intended mode. For linear data, the Lorenzo predictor reverts to a delta predictor, which processes data similarly to the first-order FSD algorithm. Hence, we use the modified *dfcm* predictor from the DFCM compressor (see above) in our implementation of PLMI, which compresses linear data better. The predicted and true floating-point values are mapped to unsigned integers 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 [28]. We evaluate all predictor sizes between 16 bytes and 512 MB that are powers of two.

5. RESULTS

This section evaluates FPC and compares it with the five compressors presented in the previous section. Section 5.1 studies the compression ratio, Section 5.2 investigates the throughput, and Section 5.3 looks at the memory consumption. Section 5.4 evaluates the predictor and Section 5.5 the critical-loop performance of FPC.

5.1 Compression Ratio

Table 2 presents the compression ratios that the six algorithms achieve on each dataset. The numbers in bold print highlight the best compression ratio for each dataset. The leftmost column lists the compression level for BZIP2 and GZIP, the order for FSD, and the binary logarithm of the number of table entries (an entry consists of two eight-byte words) for DFCM, FPC, and PLMI. To improve the readability, the table only includes results for odd DFCM, FPC, and PLMI sizes. The bottom-most row gives the compression ratio of the original PLMI algorithm (based on an executable provided by the PLMI authors), which uses the Lorenzo predictor instead of the modified *dfcm* predictor.

Table 2: Compression ratio of the six algorithms on the thirteen dataset

		message datasets					numeric datasets				observational datasets				GM
		bt	lu	sp	sppm	sweep3d	brain	comet	control	plasma	error	info	spitzer	temp	
BZIP2	1	1.102	1.021	1.075	6.783	1.061	1.039	1.145	1.028	1.383	1.295	1.095	1.287	1.021	1.290
	2	1.097	1.018	1.068	6.863	1.062	1.041	1.154	1.029	1.788	1.301	1.116	1.387	1.023	1.327
	3	1.094	1.017	1.063	6.875	1.130	1.041	1.159	1.030	2.418	1.303	1.131	1.466	1.023	1.372
	4	1.092	1.016	1.059	6.880	1.172	1.042	1.162	1.030	2.942	1.312	1.153	1.530	1.023	1.404
	5	1.091	1.017	1.056	6.878	1.190	1.042	1.165	1.030	3.523	1.321	1.153	1.584	1.023	1.430
	6	1.090	1.017	1.055	6.878	1.213	1.042	1.167	1.029	4.312	1.328	1.165	1.634	1.023	1.459
	7	1.089	1.017	1.053	6.880	1.247	1.042	1.169	1.029	4.579	1.334	1.174	1.678	1.023	1.474
	8	1.088	1.018	1.054	6.899	1.275	1.042	1.172	1.029	5.177	1.333	1.205	1.717	1.024	1.496
	9	1.088	1.018	1.055	6.933	1.294	1.043	1.173	1.029	5.789	1.339	1.217	**1.752**	1.024	1.516
DFCM	1	1.126	1.021	1.089	2.509	1.285	1.171	1.151	1.062	0.970	1.009	0.968	0.984	0.991	1.137
	3	1.162	1.134	1.138	2.705	1.283	1.171	1.138	1.053	0.969	1.098	0.968	0.986	0.999	1.167
	5	1.226	1.221	1.155	2.898	1.294	1.167	1.143	1.052	0.970	1.172	0.970	0.986	1.006	1.193
	7	1.264	1.229	1.213	3.247	1.298	1.157	1.140	1.049	0.976	1.230	0.979	0.987	1.003	1.217
	9	1.301	**1.239**	1.230	3.510	1.300	1.164	1.138	1.050	0.978	1.271	0.980	0.988	1.005	1.233
	11	1.312	1.224	1.234	3.700	1.307	1.167	1.138	1.050	0.982	1.277	0.986	0.988	1.005	1.240
	13	1.331	1.229	1.236	3.846	1.338	1.175	1.141	1.049	0.983	1.293	0.991	0.988	1.002	1.250
	15	1.351	1.233	1.241	3.948	1.383	1.194	1.148	1.053	1.303	1.321	1.217	0.991	1.007	1.312
	17	1.355	1.223	1.244	4.036	1.446	1.207	1.145	1.050	1.301	1.350	1.221	0.991	1.002	1.321
	19	1.363	1.225	1.246	4.126	1.482	1.225	1.152	1.046	1.301	1.395	1.222	0.992	1.004	1.332
	21	1.359	1.221	1.247	4.185	1.518	1.226	1.158	1.043	1.300	1.438	1.223	0.993	1.008	1.339
	23	1.362	1.223	1.248	4.215	1.553	1.230	1.166	1.045	1.301	1.495	1.225	0.994	1.011	1.349
	25	**1.363**	1.224	1.249	4.234	1.559	1.232	1.174	1.054	1.301	1.518	1.226	0.995	1.014	1.354
FPC	1	1.127	1.061	1.088	2.714	1.234	1.150	1.156	1.047	1.106	1.141	1.109	1.014	1.013	1.181
	3	1.131	1.114	1.196	3.200	1.220	1.140	1.145	1.047	1.157	1.151	1.151	1.013	1.019	1.216
	5	1.157	1.144	1.235	3.840	1.229	1.131	1.146	1.047	1.268	1.193	1.156	1.013	1.019	1.254
	7	1.185	1.142	1.247	4.198	1.233	1.130	1.147	1.049	1.312	1.229	1.138	1.013	1.017	1.270
	9	1.222	1.145	1.251	4.575	1.231	1.136	1.146	1.050	1.305	1.242	1.131	1.013	1.015	1.282
	11	1.242	1.145	1.252	4.808	1.232	1.138	1.147	1.050	1.305	1.266	1.136	1.013	1.014	1.292
	13	1.244	1.143	1.249	4.901	1.263	1.140	1.147	1.048	1.421	1.278	1.167	1.013	1.013	1.308
	15	1.255	1.157	1.247	5.082	1.604	1.153	1.147	1.046	2.687	1.324	1.255	1.013	1.010	1.418
	17	1.272	1.169	1.253	5.264	2.287	1.158	1.150	1.046	6.437	1.469	1.471	1.014	1.009	1.598
	19	1.280	1.172	1.257	5.298	2.794	1.158	1.150	1.042	11.377	1.890	1.857	1.014	1.008	1.762
	21	1.284	1.173	1.260	5.276	2.999	1.162	1.151	1.038	13.870	2.723	2.138	1.015	1.004	1.870
	23	1.287	1.172	**1.262**	5.284	3.065	1.163	1.155	1.038	14.764	3.376	2.269	1.020	0.999	1.924
	25	1.286	1.169	1.261	5.274	**3.089**	1.164	1.156	1.041	**15.048**	**3.603**	**2.270**	1.027	0.997	1.938
FSD	1	1.070	1.004	0.987	2.348	1.151	1.100	1.095	0.992	0.940	1.163	0.938	0.962	0.966	1.095
	2	1.054	0.992	0.982	2.227	1.193	1.092	1.109	0.976	0.997	1.004	1.002	0.950	0.959	1.087
	3	1.037	0.984	0.967	2.182	1.210	1.076	1.105	0.985	0.996	1.003	0.999	0.939	0.953	1.080
	4	1.022	0.986	0.946	2.139	1.195	1.058	1.089	0.968	0.979	1.001	0.985	0.937	0.969	1.069
	5	0.997	0.958	0.941	2.114	1.176	1.036	1.072	0.971	0.925	0.998	0.942	0.927	0.948	1.048
	6	0.984	0.917	0.934	2.072	1.153	1.017	1.058	0.951	0.889	0.971	0.916	0.916	0.926	1.025
	7	0.975	0.906	0.927	2.049	1.131	1.002	1.044	0.939	0.889	0.934	0.903	0.903	0.909	1.011
GZIP	1	1.127	1.050	1.108	6.314	1.085	1.057	1.157	1.054	1.592	1.422	1.146	1.219	1.034	1.323
	2	1.128	1.050	1.108	6.675	1.086	1.058	1.157	1.054	1.591	1.422	1.146	1.228	1.034	1.329
	3	1.128	1.050	1.107	6.853	1.085	1.058	1.158	1.054	1.591	1.421	1.146	1.232	1.034	1.332
	4	1.130	1.055	1.108	7.000	1.092	1.064	1.161	1.058	1.607	1.448	1.153	1.230	1.036	1.341
	5	1.130	1.055	1.108	7.217	1.092	1.064	1.161	1.058	1.608	1.448	1.154	1.231	1.036	1.344
	6	1.130	1.055	1.108	7.352	1.092	1.064	1.161	1.058	1.608	1.448	1.154	1.231	1.036	1.346
	7	1.130	1.055	1.107	7.388	1.092	1.064	1.162	1.058	1.608	1.448	1.154	1.231	1.036	1.346
	8	1.130	1.055	1.107	7.420	1.092	1.064	1.162	1.058	1.608	1.448	1.154	1.231	1.036	1.347
	9	1.130	1.055	1.107	**7.431**	1.092	1.064	1.162	1.058	1.608	1.448	1.154	1.231	1.036	1.347
PLMI	1	1.123	1.054	1.094	2.722	1.206	1.115	1.168	1.057	1.133	1.054	1.138	1.067	1.033	1.182
	3	1.133	1.123	1.134	2.918	1.207	1.116	1.164	1.057	1.139	1.113	1.142	1.067	1.034	1.204
	5	1.162	1.181	1.161	3.210	1.207	1.117	1.169	1.060	1.149	1.173	1.147	1.069	1.035	1.229
	7	1.189	1.184	1.175	3.598	1.207	1.115	1.171	1.060	1.137	1.197	1.143	1.070	1.035	1.244
	9	1.214	1.188	1.178	3.908	1.207	1.116	1.176	1.061	1.150	1.224	1.144	1.071	1.035	1.259
	11	1.223	1.182	1.180	4.208	1.208	1.115	1.177	1.061	1.143	1.229	1.142	1.072	1.035	1.266
	13	1.234	1.184	1.182	4.383	1.208	1.117	1.180	1.060	1.183	1.233	1.143	1.073	1.035	1.276
	15	1.238	1.185	1.182	4.495	1.209	1.119	1.182	1.061	1.232	1.239	1.148	1.074	1.035	1.284
	17	1.239	1.181	1.184	4.719	1.210	1.121	1.181	1.061	1.243	1.239	1.151	1.075	1.035	1.290
	19	1.240	1.181	1.184	4.817	1.211	1.123	1.183	1.062	1.244	1.245	1.152	1.076	1.035	1.293
	21	1.239	1.180	1.185	4.962	1.212	1.123	1.182	1.062	1.255	1.249	1.152	1.078	1.036	1.298
	23	1.240	1.180	1.186	5.003	1.212	1.125	1.183	1.062	1.255	1.261	1.155	1.079	1.036	1.300
	25	1.241	1.180	1.186	5.025	1.213	1.125	1.184	1.063	1.255	1.262	1.156	1.081	1.037	1.302
PLMI		*1.200*	*1.134*	*1.112*	*3.249*	*1.332*	***1.245***	***1.265***	***1.124***	*1.063*	*1.365*	*1.056*	*1.075*	***1.088***	*1.263*

With table sizes above one megabyte, FPC achieves the highest geometric-mean compression ratio. It outperforms the other five algorithms by a large margin on four datasets. However, on the two datasets *msg_sppm* and *obs_spitzer*, GZIP and BZIP2 substantially outperform FPC, respectively. Surprisingly, BZIP2 delivers the second highest geometric-mean compression ratio even though it was not specifically designed for compressing floating-point data.

DFCM is superior to FPC in some instances because it employs a more sophisticated predictor, which stores two difference values in each table entry and uses a more elaborate hash function. However, FPC outperforms DFCM on the majority of our datasets because FPC contains two predictors that complement each other, i.e., when one of them performs poorly, the other often performs well (cf. Section 5.4).

The original version of PLMI outperforms our modified PLMI version on six datasets and excels over all other algorithms on four datasets. Interestingly, all of these datasets are generally poorly compressible. On average, the original PLMI algorithm performs at the level of our version with 1024 table entries (i.e., 16 kilobytes of state).

DFCM occasionally outperforms PLMI (e.g., on *msg_sweep3d* and *obs_error*). Both algorithms employ the same predictor and, intuitively, the fixed-length codes used by DFCM should be inferior to PLMI's variable-length codes because of the nonuniform symbol distribution (cf. Section 5.4). However, DFCM's coding scheme is sometimes superior for two reasons. First, PLMI uses a quasi-static probability model to estimate the symbol distribution on the fly. The estimate is solely based on the frequency of the previously seen symbols. If the estimated distribution does not accurately reflect the distribution of the following symbols, it can result in suboptimal code lengths. Second, PLMI employs range coding that outputs data in byte increments. Consequently, even though variable-length coding is employed, the algorithm often needs to insert padding bits to stay at a byte boundary. Our measurements show that on average 3.46 bits of padding are added per double with a one-million-entry predictor.

No compression algorithm performs best on more than five of the thirteen datasets. There is also no best algorithm within the three dataset categories. Even BZIP2 and GZIP, the general-purpose compressors, provide the highest compression ratio in some cases.

With the exception of *msg_sppm*, which can be compressed by at least a factor of two, none of our datasets are highly compressible with the algorithms we studied. All six algorithms are

ineffective on *num_control* and *obs_temp*, which they compress by no more than six and four percent, respectively. These results are consistent with the randomness information from 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 is obtained on *num_plasma*, which contains the lowest fraction of unique values.

Some datasets, most notably *msg_sweep3d*, *num_plasma*, *obs_error*, *obs_info*, *obs_spitzer*, and *msg_sppm* much prefer one algorithm over the others. (For example, *num_plasma* strongly favors FPC, which compresses this dataset by more than a factor of fifteen, over the other algorithms, which compress it less than half as much.) With the exception of *msg_sweep3d*, these datasets contain relatively few unique values.

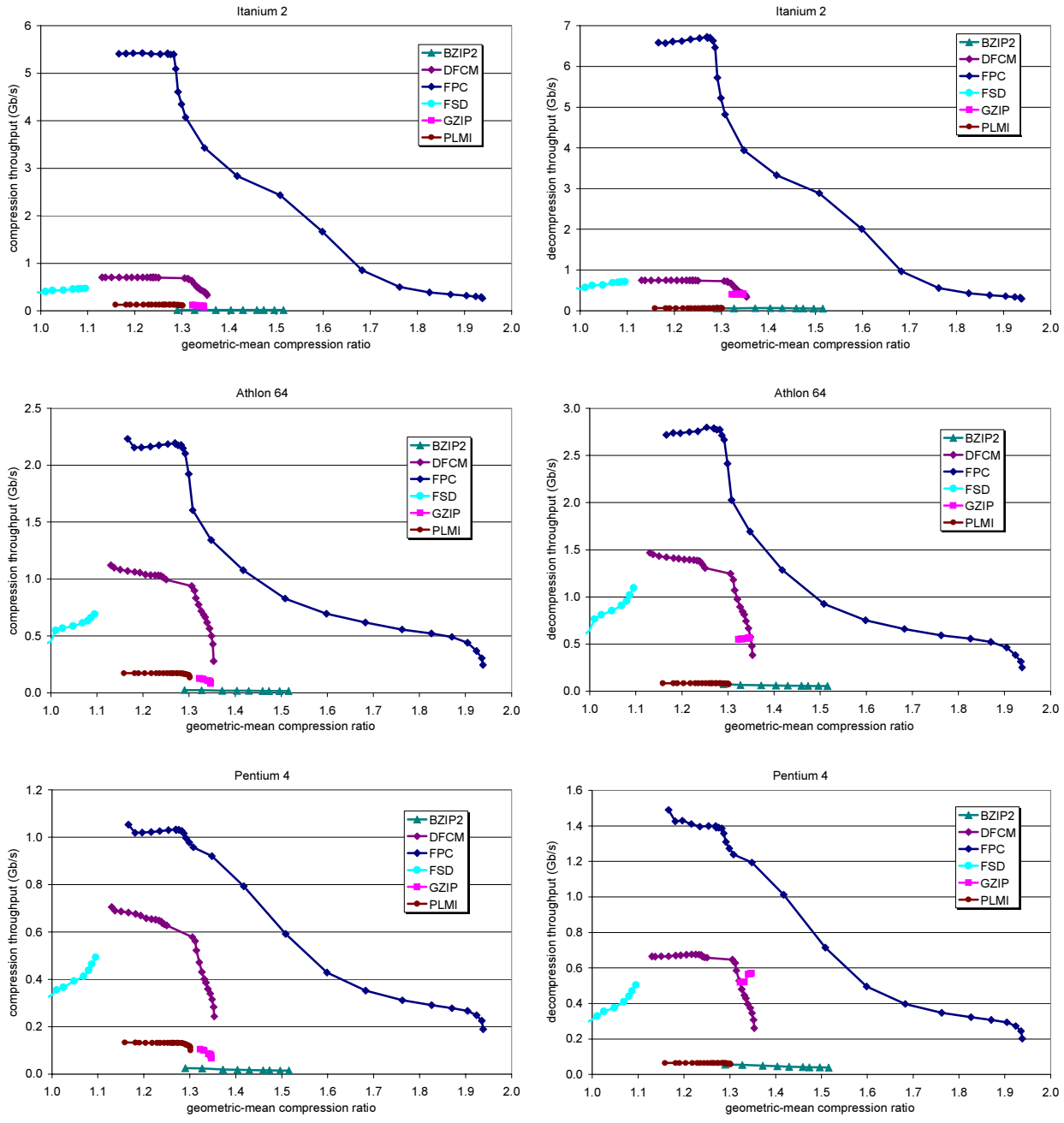
The five datasets with above 99.5% randomness (*msg_lu*, *msg_sp*, *num_brain*, *num_control*, and *obs_temp*) cannot be compressed by more than 26% by any of the algorithms we studied.

Increasing the level (i.e., the block size) of BZIP2 increases the compression ratio by more than three percent on four datasets and hurts the performance on *msg_bt* and *msg_sp*. Increasing the level of GZIP boosts the compression ratio by more than two percent only on *msg_sppm*. FSD performs worse with higher orders on our datasets. There are some cases where orders two, three, or four are best, but most of the time order one results in the highest compression ratio. Increasing the predictor size improves the compression ratio by more than ten percent on eight datasets for DFCM and FPC and on five datasets for PLMI. However, FPC's performance decreases with larger table sizes on *num_control* and *obs_temp* (i.e., the two datasets that are the hardest to compress). The same is true for DFCM on *num_control*. In summary, increasing the predictor size or level is only worthwhile on some datasets, usually the ones that are easier to compress. Moreover, as we shall see next, such an increase comes at the cost of decreased throughput.

5.2 Throughput

This section examines the compression and decompression throughput of the six algorithms (i.e., the raw dataset size divided by the runtime). Figure 2 presents the results from the four systems described in Section 4.1. Each panel plots the throughput in gigabits per second versus the geometric-mean compression ratio. The four rows of panels correspond to the four platforms. The

left panels show the compression and the right panels the decompression results. For DFCM, FPC, and PLMI, the predictor table size doubles for each data point from sixteen bytes (leftmost) to 512 MB (rightmost). For BZIP2 and GZIP, the individual data points correspond to levels one (leftmost) through nine (rightmost). For FSD, the figure shows results for order one (rightmost) through order seven (leftmost). Note that the y-axes are scaled differently in each panel to improve readability.



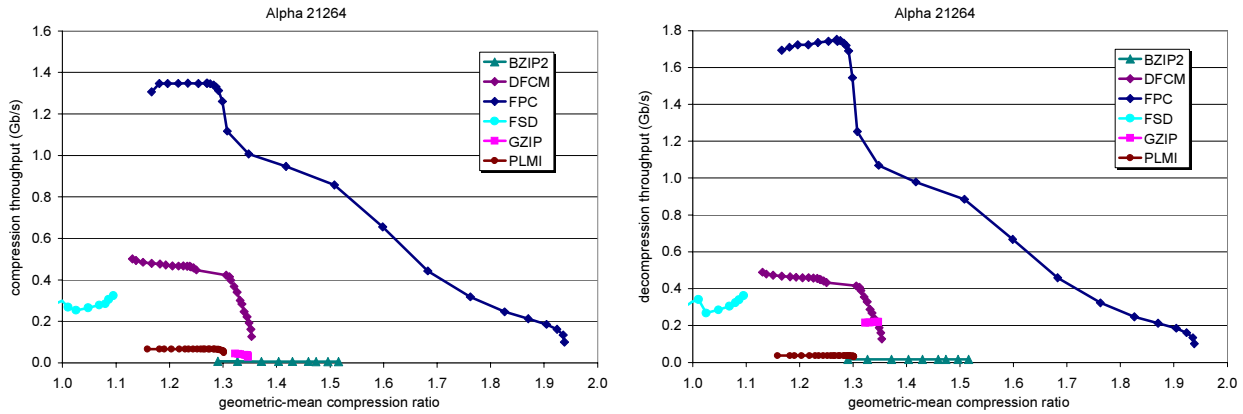


Figure 2: Geometric-mean compression throughput (left panels) and decompression throughput (right panels) versus compression ratio of the six algorithms on the four systems

For a given compression ratio, FPC exceeds the throughput of the other algorithms by a large margin on all four architectures. DFCM has the second highest throughput, though, occasionally (e.g., on the Pentium 4), GZIP's decompression throughput is a little higher. 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. On the Itanium 2, FPC compresses our datasets 8 to 300 times faster and decompresses them 9 to 100 times faster than the other algorithms at the same geometric-mean compression ratio. It reaches a compression throughput of up to 5.43 Gb/s and a decompression throughput of up to 6.73 Gb/s.

FSD's compression and decompression throughput is roughly half a gigabit per second, except on the Athlon 64, where it decompresses at over one gigabit per second. DFCM also performs best on the Athlon 64, where it reaches about 1.1 to 1.5 Gb/s compared to well under one gigabit per second on the other three machines. PLMI's highest throughput is below 0.2 Gb/s on all machines, but it, too, is the highest on the Athlon 64. These three algorithms seem to benefit from the Athlon's combination of a high clock speed, large L1 data cache, and 64-bit support. Although the Pentium 4 has an even higher clock speed, it lacks 64-bit support, provides fewer logical registers, and has a smaller L1 cache.

GZIP runs faster on the two x86 machines than on the Alpha and the Itanium. Compression is slightly faster on the Athlon 64 whereas decompression is slightly faster on the Pentium 4. GZIP

operates at byte granularity and is therefore not sensitive to 64-bit support. The same is true for BZIP2. However, BZIP2's memory footprint is substantially larger than that of GZIP (cf. Section 5.3), which is probably why it is more sensitive to the cache size. At least for decompression, its throughput on the Itanium 2 is significantly higher than on the Pentium 4, the latter of which is clocked almost twice as fast (internally almost four times as fast) but has a smaller cache hierarchy.

The Athlon 64 system delivers the highest compression throughput on all algorithms except FPC, where the Itanium 2 is faster. The Itanium 2 system is otherwise the second fastest, followed by the Pentium 4, which is outperformed on FPC by the Alpha 21264, the otherwise slowest system. The same observations apply to the decompression throughput.

The Athlon 64 has the highest clock speed of the three 64-bit systems and the largest L1 data cache (together with the Alpha 21264). This combination of features makes it the system of choice for most of the compression algorithms we studied. FPC prefers the Itanium 2 because this processor provides the largest number of logical registers and the highest internal parallelism (cf. Section 5.5). These two reasons, together with the fact that the Pentium 4 is a 32-bit CPU, also explain why FPC runs faster on the Alpha than on the Pentium 4.

Figure 3 combines the FPC throughput results from the four systems in one graph. It plots the throughput in millions of doubles per second against the binary logarithm of the number of hash-table entries.

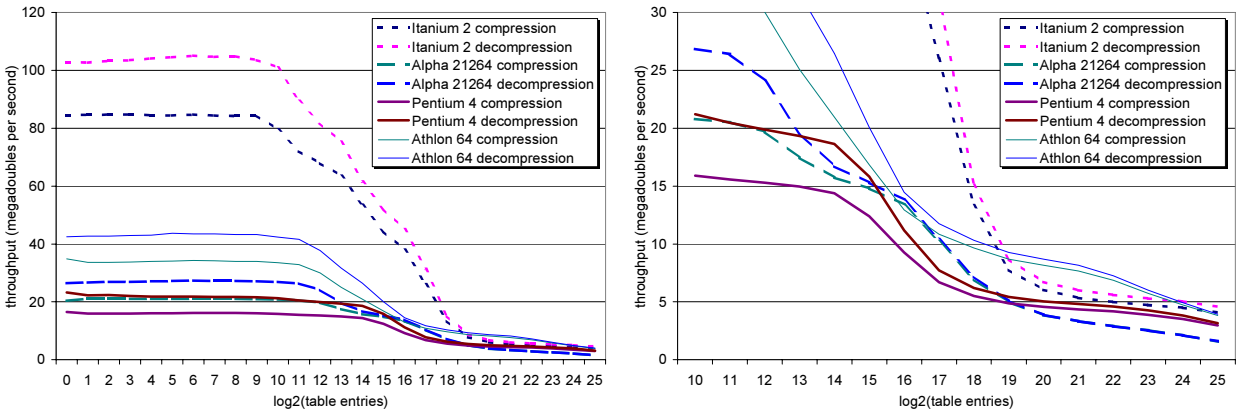


Figure 3: FPC's compression and decompression throughput versus hash-table size on the four systems (the right panel shows a zoomed in version of the left panel's lower right corner)

FPC runs over twice as fast on the Itanium 2 as on the other three machines for table sizes up to about four megabytes (the L3 cache has a capacity of three megabytes). As mentioned, the two most important reasons for this performance difference are that the Itanium 2 has the highest issue width of the four CPUs (it can sustain 6 as opposed to 4 or 3 executed instructions per cycle) and that it has the most logical general-purpose registers (128 as opposed to the Alpha's 32, the Athlon's 16, and the Pentium's 8) and therefore does not spill any scalar variables or temporaries.

The Pentium 4 performs the worst for table sizes up to a few megabytes. This is mostly because it is the only 32-bit machine we studied (our FPC implementation uses almost exclusively 64-bit operations) and it has only eight logical general-purpose registers. As a consequence, the Pentium 4 has to spill and fill registers all the time. Moreover, it needs to execute at least two machine instructions for every 64-bit operation for which the other three machines only require one instruction. Evidently, the higher clock speed is not able to compensate for this overhead. The Athlon 64 has fewer registers (16) than the Alpha 21264 (32) but outperforms it because its clock speed is almost 2.5 times higher.

At the high end, i.e., up to half a gigabyte of table space, FPC frequently misses in all cache levels due to the rather random hash-table accesses. Hence, the CPU parameters do not matter much and the memory controller and the main memory latency largely determine the performance. In this range, we find the Athlon 64 system to dominate except for the largest two table sizes, where the Itanium 2 takes the lead again. We suspect the Athlon's superior performance to be the result of the integration of the memory controller with the core on the same die, which significantly reduces the access latency. At the largest size we measured (half a gigabyte), the performance is mostly determined by the throughput of the main memory, which is higher on the Itanium 2 server than on the Athlon 64 workstation. The Pentium 4's memory system is designed for 32-bit operations and has to handle twice the number of accesses, which is why it is slower. The Alpha 21264 server has the slowest memory system, probably because it is over four years older than the other three machines.

5.3 Memory Consumption

This section studies the memory footprint, as reported by the UNIX command *ps*, of the six algorithms. Figure 4 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 4 plots the larger of the two amounts. The individual datapoints in the figure again correspond to different table sizes, levels, and orders. The results were measured on the Itanium 2 system, but the trends are the same on the other three systems.

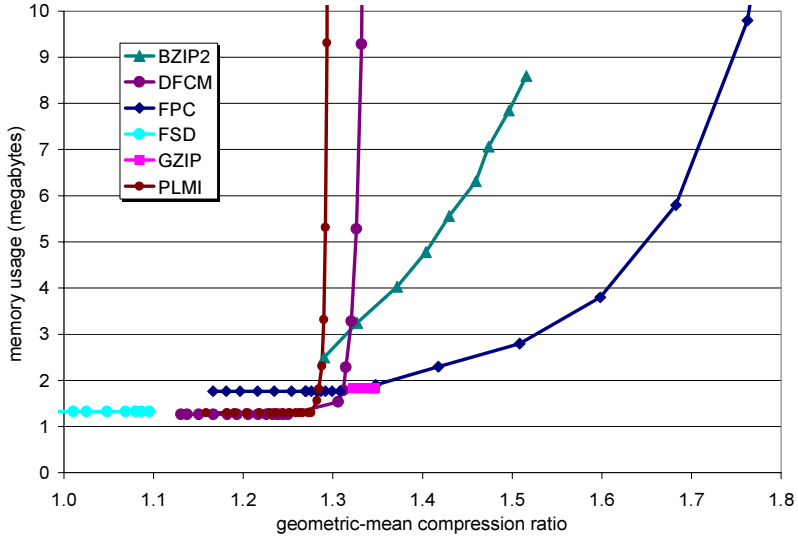


Figure 4: Memory usage versus compression ratio of the six algorithms on the Itanium 2 system

Except for FPC, all algorithms basically reach their highest geometric-mean compression ratio with less than ten megabytes of memory. In fact, the benefits are already small above two megabytes for most of the compressors. 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, the code and stack size as well as the input and output buffers determine FPC's memory usage. But for larger sizes, the two predictor tables dominate, as can be seen from the exponentially growing curve. The same is true for DFCM and PLMI. However, unlike their modified *dfcm* predictor, FPC's two predictors can effectively turn additional memory (up

to about ten megabytes) into higher compression ratios. The next 500 megabytes yield less than a ten percent increase in compression ratio for FPC.

5.4 Predictor Usage

Figure 5 shows how often the *fcm* and the *dfcm* predictions are used in FPC, i.e., how often they result in more leading zero bytes than the prediction of the other predictor. A pair of bars is shown for each dataset; one bar for 1024 hash-table entries and the other for 1,048,576 entries.

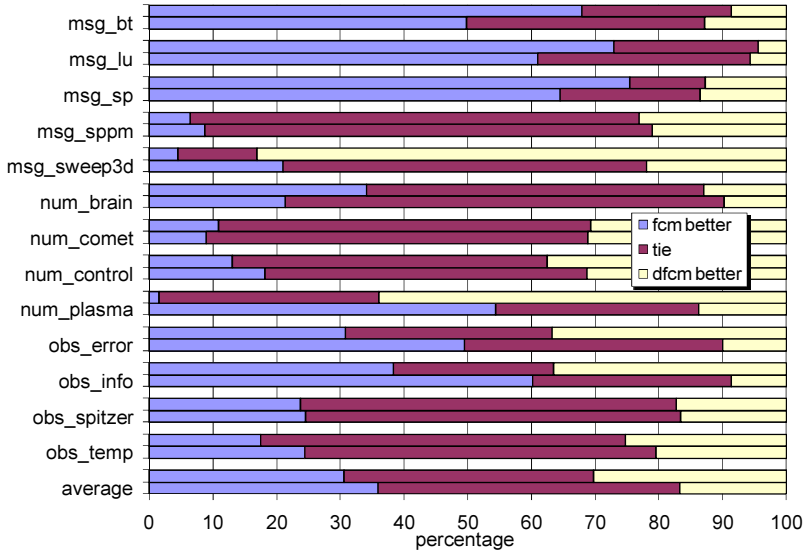


Figure 5: Percentage of time the *fcm* and *dfcm* predictions provide more leading zero bytes (in each pair, the top bar corresponds to 1,048,576-entry and the bottom bar to 1024-entry tables)

We find that the two predictors complement each other well. Only on *msg_lu* and *msg_sppm* is one predictor needed less than ten percent of the time with the small and the large table size. In other words, both predictors are used frequently in most cases.

Some datasets result in rather biased usage. On the one hand, *fcm* is useless 83.1% of the time on *msg_sweep3d* with one million entries. On the other hand, *fcm* yields more leading zero bytes 75.5% of the time on *msg_sp* with one million entries. These results highlight the importance of having more than one predictor and explain why FPC compresses many datasets better than the related DFCM algorithm, which only uses a single predictor.

On some datasets, e.g., *obs_spitzer* and *msg_sppm*, the frequency of usage does not change much when changing the table size. On other datasets, e.g., *msg_sweep3d* and *num_plasma*, the frequency changes by a large amount. Interestingly, the datasets that result in different usage frequencies typically see significant improvements in their compression ratios due to the much better performance of the *dfcm* predictor with larger tables (cf. Table 2).

Figure 6 shows the distribution of the number of leading zero bytes after xoring the true double values with the more accurate of the two predictions. The results are averages over the thirteen datasets.

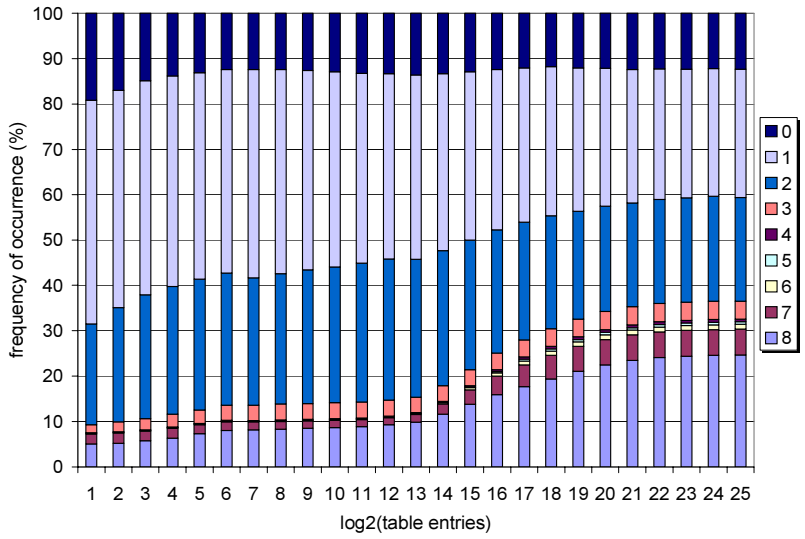


Figure 6: Average distribution of the leading zero byte counts for different table sizes

With table sizes of more than eight entries, less than 15% of the doubles result in no leading zero bytes. Because of the added 4-bit code in the compressed output, these values are expanded instead of compressed by FPC. However, the other 85% of the values are compressed. One and two leading zero bytes are common for all predictor sizes, but eight zero bytes (i.e., all 64 bits predicted correctly) are also frequent with larger hash tables. Seven and three leading zero bytes occur less than 7% of the time and six, five, and four leading zero bytes are very infrequent (<1.1%).

5.5 Critical-Loop Performance

This section studies the critical loop in the compression and in the decompression function of our FPC implementation. These two loops compress and decompress one block of data (comprising up to 32,768 doubles), respectively. For hash tables that fit in the L1 data cache, a little over 90% of the compression time is spent in the critical loop and basically all of the remaining time is spent moving data into and out of the buffers in I/O operations. Similarly, just under 90% of the decompression time is spent in the critical loop with the rest of the time going to data movement. With larger hash table sizes, the loops run more slowly because of cache misses and the percentage of the total runtime they represent increases.

We investigated the assembly listing of the two loops on each of our four systems. Table 3 shows the number of static machine instructions in the loop bodies as well as the approximate average number of static instructions in the loops needed to compress and decompress one double.

Table 3: Static instruction count of the critical loops in FPC

	compression		decompression	
	loop body	per double	loop body	per double
Itanium 2	138	69.0	93	46.5
Athlon 64	185	92.5	110	55.0
Pentium 4	367	183.5	361	120.3
Alpha 21264	121	60.5	253	42.2

The results indicate that FPC requires roughly 60 to 90 machine instructions (i.e., operations) on 64-bit CPUs to compress a double and roughly 40 to 55 instructions to decompress a double. The 32-bit CPU is over a factor of two less efficient because the compiler has to synthesize 64-bit operations out of 32-bit instructions. Moreover, due to the small number of logical registers, a large amount of spill and fill code is needed for the Pentium 4.

The Alpha compiler emits the entire loop as a single basic block (all IF statements are converted into conditional move instructions) and uses the fewest instructions per double. It is the only compiler that chose to unroll one of the loops (it unrolled the decompression loop three times). Both loops contain one NOP for instruction slotting purposes [8]. The decompression loop contains a software prefetch instruction.

The Itanium compiler generates slightly more instructions per double, partially because both loops contain four NOPs due to static scheduling constraints [19]. Each loop contains a software prefetch instruction. The loops are also single basic blocks and are software pipelined. Because the Itanium 2 is an in-order, statically scheduled machine, we can determine the number of cycles a loop iteration takes assuming there are no L1 cache misses. A compression iteration takes 25 cycles (12.5 cycles per double) and a decompression iteration takes 18 cycles (9 cycles per double). This means that the compression loop executes an average of 5.5 instructions per cycle and the decompression loop 5.16 instructions per cycle. This very high ILP is quite close to the CPU's maximum of six executed instructions per cycle. Moreover, it is substantially higher than the fetch width of the other three machines we studied, which explains why FPC runs particularly well on the Itanium 2.

The fact that the two critical loop bodies are single basic blocks has an important implication. The exact same sequence of instructions is executed to compress/decompress a block of doubles regardless of the data values or their compressibility. The running time of these loops, which account for most of the total runtime, is therefore only dependent on the load latency, as all other instructions have fixed latencies. In other words, as long as the hash tables fit in the L1 data cache, the compression and the decompression time for a block of data are constant no matter what data are being processed. This rather unusual feature, which most other compression algorithms do not possess, is a requirement in real-time environments.

The Athlon requires noticeably more instructions to express the loop bodies, mostly because of a significant number of register spills and fills. More than 16 (but no more than 32) registers are needed to hold all the variables and temporaries. While the decompression loop is also a single basic block, the Athlon compiler only converts 16 of the 18 IF statements in the compression loop into conditional moves and emits two conditional jumps. The 16 converted IF statements all compare a value to zero and have only a single assignment in their bodies. The remaining two IF statements are more complex.

The Pentium code of the compression loop includes the same two conditional branches. Moreover, almost exactly twice as many instructions are emitted for the 32-bit x86 CPU as for the 64-bit x86 CPU. The decompression code consists of multiple basic blocks that are emitted out of program order, i.e., the loop body is not straight-line code but consists of chunks of code that jump to other chunks. Moreover, about half of the code is duplicated in the various chunks.

6. CONCLUSIONS

This paper describes FPC, a lossless compression algorithm for linear streams of double-precision floating-point values. It uses two context-based predictors to sequentially predict each value in the stream. The prediction and the true value are xored and the xor result is leading zero byte compressed. This algorithm features a high speed, good compression ratio, and ease of implementation. In addition, 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 predictor tables that fit into the L1 data cache, it delivers a guaranteed throughput of over 84 million doubles per second on a 1.6 GHz Itanium 2. This corresponds to only two machine cycles to process a byte of data. The source code, a line by line description thereof, and the datasets are available at http://www.csl.cornell.edu/~burtscher/research/FPC/.

The current version of FPC does not compress structured datasets (e.g., multidimensional datasets), 32-bit floating-point values, and easy-to-compress data particularly well. Hence, we want to generalize FPC by adding support for exploiting structure, designing a version that is optimized for single-precision data, and including an optional second compression stage. To further improve the speed of FPC, we are planning on writing a parallel version. We also intend to look at compressing each block independently (i.e., zero out the predictor state instead of continuing with the state from the previous block). Doing so enables the simultaneous compression/decompression of multiple blocks and allows fast-forwarding without the need to decompress all preceding values. Preliminary experiments show that compressing blocks of several kilobytes independently reduces the compression ratio by no more than a couple of percent with small to medium predictor sizes.

7. ACKNOWLEDGEMENTS

The authors and this project are supported by the Department of Energy under Award Number DE-FG02-06ER25722. Martin Burtscher is further supported by NSF grants CNS-0724966 and CNS-0719966 as well as grants from IBM and Intel. Intel donated some of the machines we used to evaluate the compressors. The views and opinions expressed herein do not necessarily state or

reflect those of the DOE, IBM, Intel, or the NSF. Drs. Peter Lindstrom and Martin Isenburg provided the executable of the original PLMI algorithm. 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*. One of the authors generated *num_brain* using a modified version of EULAG [5], [25], 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 five 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." *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*. May 1994.

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

[4] M. Burtscher and P. Ratanaworabhan. "High Throughput Compression of Double-Precision Floating-Point Data." *2007 Data Compression Conference*, pp. 293-302. March 2007.

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

[6] M. Burtscher and B. G. Zorn. "Exploring Last *n* Value Prediction." *International Conference on Parallel Architectures and Compilation Techniques*, pp. 66-76. October 1999.

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

[8] Compaq. "Compiler Writer's Guide for the Alpha 21264." *ftp://ftp.digital.com/pub/Digital/info/semiconductor/literature/dsc-library.html*. June 1999.

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

[10] J. Fowler and R. Yagel. "Lossless Compression of Volume Data." *IEEE Symposium on Volume Visualization*, pp. 43-50. 1994.

[11] F. Gabbay. *Speculative Execution Based on Value Prediction*. EE Department Technical Report #1080, Technion - Israel Institute of Technology. November 1996.

[12] 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.

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

[14] 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*, pp. 207-216. January 2001.

[15] http://www.bzip.org/, 2007.

[16] http://www.gzip.org/, 2007.

[17] http://www.llnl.gov/asc/computing_resources/purple/rfp/benchmarks/limited/code_list.html, 2007.

[18] 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.

[19] Intel Corporation. “Intel Itanium 2 Processor Reference Manual for Software Development and Optimization.” *http://www.intel.com/design/itanium2/manuals/251110.htm*. May 2004.

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

[21] 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. November 2004.

[22] 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.

[23] P. Lindstrom and M. Isenburg. “Fast and Efficient Compression of Floating-Point Data.” *IEEE Transactions on Visualization and Computer Graphics*, Vol. 12, No. 5. September 2006.

[24] 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*, pp. 138-147. October 1996.

[25] 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.

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

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

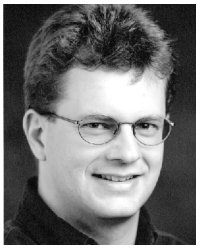
[28] M. Schindler. “A Fast Renormalisation for Arithmetic Coding.” *Data Compression Conference*, p. 572. March 1998.

[29] C. Touma and C. Gotsman. “Triangle Mesh Compression.” *Graphics Interface*, pp. 26-34. 1998.

[30] 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. October 1996.

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

[32] 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.



Martin Burtscher received the combined BS/MS degree in computer science from the Swiss Federal Institute of Technology (ETH) Zurich in 1996 and the Ph.D. degree in computer science from the University of Colorado at Boulder in 2000. He was an assistant professor in the School of Electrical and Computer Engineering at Cornell University until 2007. Since then, he has been a research scientist in the Center for Grid and Distributed Computing at the University of Texas at Austin. His research interests include automatic parallelization and optimization of irregular programs, high-speed data compression, hardware- and software-based prefetching, and brain injury simulation. He is a senior member of the IEEE, its Computer Society, and the ACM.

Paruj Ratanaworabhan received a BENG and an MENG degree in Electrical Engineering from Kasetsart University and Cornell University, respectively. He is now working on his Ph.D. in Electrical and Computer Engineering at Cornell University, where he is a member of the Computer Systems Laboratory. His research encompasses race detection and toleration, phase-aware computer architectures, data compression, and compiler optimizations. Currently, he is a visiting student at the Center for Distributed and Grid Computing at the University of Texas at Austin.