# A Multi-Phase Approach to Floating-Point Compression

Kevin R. Townsend, Joseph Zambreno
Department of Electrical and Computer Engineering
Iowa State University
Ames, IA, USA
ktown@iastate.edu, zambreno@iastate.edu

***Abstract*—This paper presents a lossless double-precision floating point compression algorithm. Floating point compression can reduce the cost of storing and transmitting large amounts of data associated with big data problems. A previous algorithm called FPC performs well and uses predictors. However, predictors have limitations. Our program (fzip) overcomes some of these limitations. fzip has 2 phases, first BWT compression, second value and prefix compression with variable length arithmetic encoding. This approach has the advantage that the phases work together and each phase compresses a different type of pattern. On average fzip achieves a 20% higher compression ratio than other algorithms.**

## I. Introduction

This paper deals with lossless double-precision floating point compression. This involves compressing an array of floating point values to take up less space in an archive. Similarly, there has to exist a way to decompress this archive back into the original data.

Floating point compression has multiple uses. In scientific computing, floating point compression has improved the performance of parallel computing applications. Many-core computers have increased in computation capacity, but internal communication has not increased as fast. However, compression has increased the performance of MPI implementations [1]. Also, floating point compression has accelerated RAM access in applications like SpMV (Sparse Matrix Vector Multiplication) [2], [3].

Current general lossless compression applications, like gzip, perform well, however, general compression algorithms can not always achieve good compression [4]. Most floating point datasets have characteristics that provide opportunities for good compression. For example, many datasets contain repeated values [2], [5].

Fig. 1 shows an analysis of repeated values. General compression schemes like gzip and algorithms specific to floating point compression like FPC do not single out this particular feature. In contrast, we single out this feature for better compression.

Take the following simple scheme: Store the repeated values separately from the rest of the data stream. With this scheme an index replaces each repeated value in the original data stream. This simple scheme works remarkably well for some datasets. However, we can achieve better compression. For example, when many repeating sequences of values exist, compressing them can vastly outperform the previous compression scheme. In total, fzip takes advantage of three compressible features of datasets: repeating sequences (more than 8 bytes long), repeating values (8 bytes long) and repeating prefixes (less than 8 bytes long).

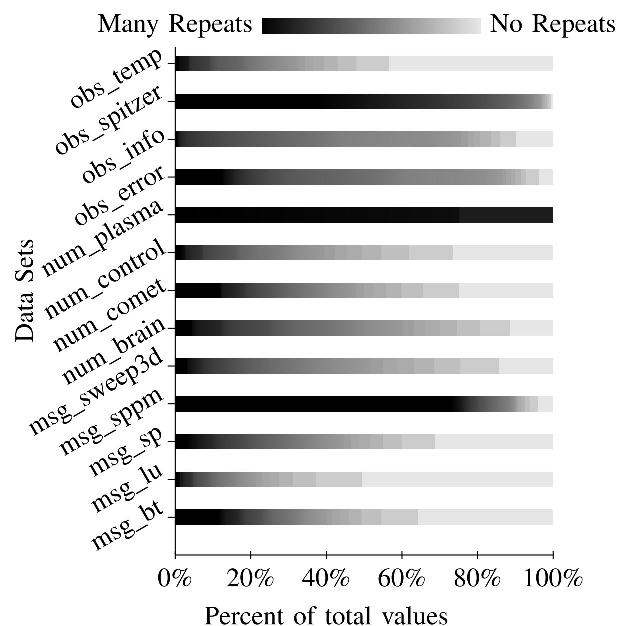



Fig. 1: The above figure shows the distribution of repeats in each dataset. Each shade represents a different number of repeats. For instance: ■:> 512, ■:16, ■:2, ■:1(no repeats).

In the remainder of this paper we talk about previous approaches (Section II), an analysis of floating point datasets (Section III), our approach to floating point compression (Section IV) and our results (Section V).

## II. Related Work

FPC, the program most similar to our work, uses predictors to compress data. A predictor uses the previous data in the data set to guess the current element in the data set. FPC uses hash functions to implement their predictors. Passing an argument between 1 and 25 to FPC configures the hash table size. Larger hash tables necessarily predict better because values in the

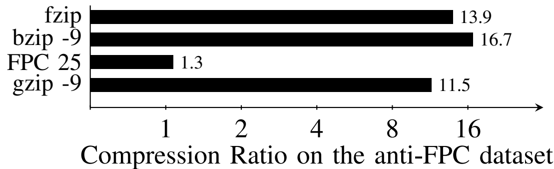



Fig. 2: We engineering a dataset to make the performance of FPC look bad compared to other programs. Although unfair, this shows a type of pattern that FPC does not exploit and other programs do. This problem exists because FPC only uses predictors for compression.

hash table get overwritten less often. However, large hash tables cause slower performance compared to hash tables that can fit in cache.

In FPC, each value gets encoded with a 4 bit header followed by 0, 1, 2, 3, 4, 6, 7 or 8 bytes. The first bit specifies which of the 2 predictors resulted in the most matching bytes compared to the correct current value. Then, the next 3 bits encode this number of bytes, however, 5 bytes is rounded down to 4 bytes. Then, the least significant bytes that the hash failed to predict follow.

FPC has the advantage of speed. Particularly when the hash table fits in cache. Although good, the compression ratio suffers because predictors do not always get the best or a good prediction. If a pattern does not exist among the values in the data set then FPC can not predict with any accuracy. Let us design this "anti-FPC" dataset. Take the set of numbers $\{10^0, 10^1, 10^2, 10^3, \ldots, 10^9\}$ and randomly choose one (with replacement) to add to the anti-FPC dataset. We do this a million times to get a dataset with a million values (8MB in size). You can observe the performance of this in Fig. 2.

## III. Floating-Point Value Analysis

Continuing the analysis from Section I, Fig. 1 shows an analysis of the repeating values in each of the datasets used for testing. Several characteristics of this analysis suggest that compressing repeating values will perform well. For example, in half of the datasets at least 80% of the values repeat.

Another pattern exists among the prefixes of the values. To understand why, look at the floating point data structure. Double-precision floating-point values have 3 parts: a sign bit, 11 exponent bits and 52 fraction bits. Values close to each other in the dataset often share the same sign. (Some datasets only contain positive numbers.) Likewise, close values often share the most significant bits of the exponent. In fact, the bits in floating-point values already exist in most likely shared to least likely shared sorted order: {sign bit, most significant exponent bits, least significant exponent bits, most significant fraction bits, least significant fraction bits}.

We gauge the strength of the pattern in a particular dataset by looking at how many prefix bits the adjacent values share. Fig. 3 describes this analysis. From this figure, we see that the first byte or so often repeats. However, there usually exists a

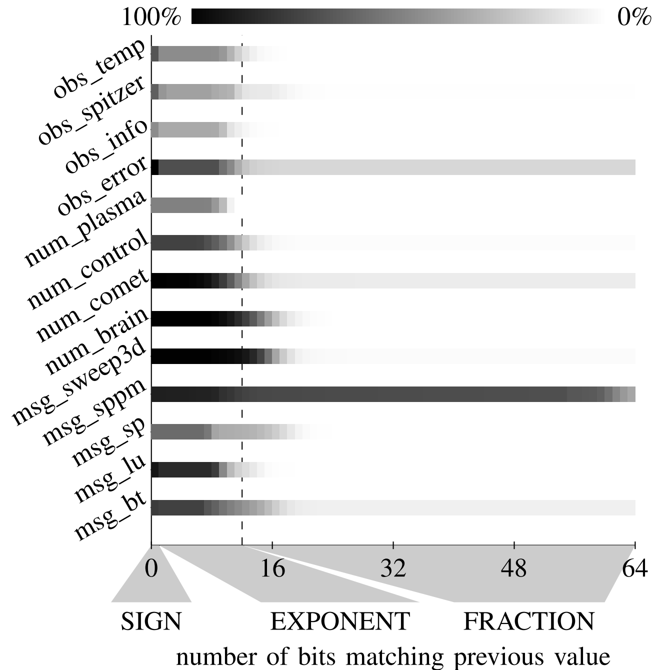



Fig. 3: The above figure represents local prefix prediction. The figure shows the density function of 2 adjacent values sharing at least $x$ number prefix bits. All of the data sets start at $(0, 100\%)$. The curves end at the percent of values that are identical to their previous value for that dataset.

rapid decline in shared bits after this point.

Datasets might also have repeating patterns of values. For example, the sequence $1.0, 2.0, 3.0, 1.0, 2.0, 3.0$ has an obvious pattern. One can use the Burrows-Wheeler Transform [6] to analyze these patterns. Fig. 4 describes this algorithm some, however, many other sources describe this algorithm in more detail [4], [6]. Fig. 5 analyzes the number of repeats that appear after the Burrow-Wheeler Transform. As the figure shows, 4 of the 13 test cases have a lot of patterns, but the rest have relatively few.

## IV. Our Approach

Our approach takes advantage of the features in Section III. The algorithm starts with BWT compression, then compresses further with using prefix and value compression.

### A. Burrows-Wheeler Transform Compression

After completing the Burrows-Wheeler Transform, fzip uses a simple encoding scheme (Fig. 4c). For each value in the BWT, fzip pushes a '0' or '1' onto a bit array to denote whether the value equals the previous value. If the values differ (a '0' in the bit array) a second array stores the next value. The 4 datasets (msg_sppm, num_plasma, obs_error, obs_info) expected to do well from the analysis in Fig. 4 do perform well under this compression scheme (see Fig. 6).

ABCDEABCDEABC$
$ABCDEABCDEABC
C$ABCDEABCDEAB
BC$ABCDEABCDEA
ABC$ABCDEABCDE
EABC$ABCDEABCD
DEABC$ABCDEABC
CDEABC$ABCDEAB
BCDEABC$ABCDEA
ABCDEABC$ABCDE
EABCDEABC$ABCD
DEABCDEABC$ABC
CDEABCDEABC$AB
BCDEABCDEABC$A

(a) Step1: Generate every cyclic rotation of the original sequence (the first row).

ABCDEABCDEABC$
ABCDEABC$ABCDE
ABC$ABCDEABCDE
BCDEABCDEABC$A
BCDEABC$ABCDEA
BC$ABCDEABCDEA
CDEABCDEABC$AB
CDEABC$ABCDEAB
C$ABCDEABCDEAB
DEABCDEABC$ABC
DEABC$ABCDEABC
EABCDEABC$ABCD
EABC$ABCDEABCD
$ABCDEABCDEABC

(b) Step2: Sort the rotations. Then the last element in each new row creates the transformed sequence.

$EEAAABBBCCDDC
New arrays:
11010010010101
$EABCDC

(c) Step3: This transformed sequence often has consecutive repeats. (This example does.) Then compression occurs by storing the first element of each repeating subsequence and storing whether the previous element equaled the current value.

Fig. 4: Above shows the Burrows-Wheeler Transform and subsequent compression. Steps 1 and 2 show the brute force calculation of BWT. Step 3 shows the basic compression used in fzip.

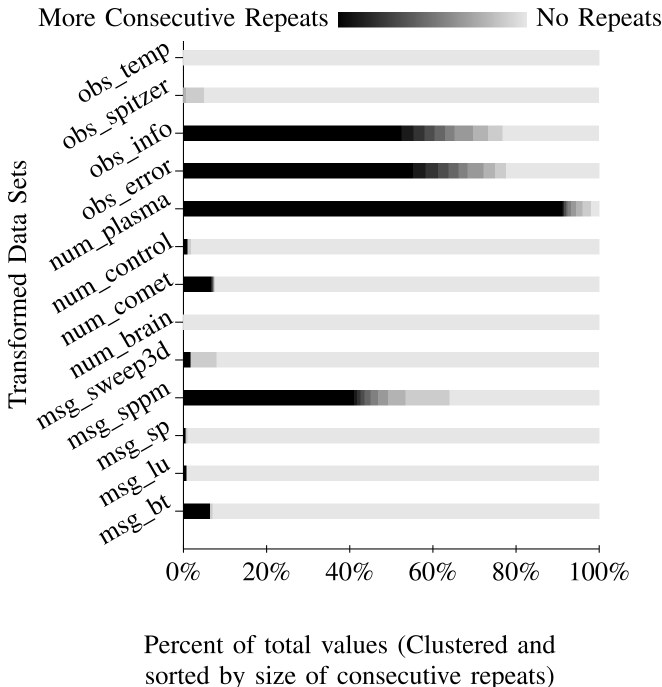



Fig. 5: Pattern analysis using the Burrows-Wheeler Transform. Each shade represents the number of consecutive repeats in a repeating sequence. ■ represents sequences longer than 9. ■ represents sequences of length 5. ■ represents sequences equal to 1 (non-repeating).

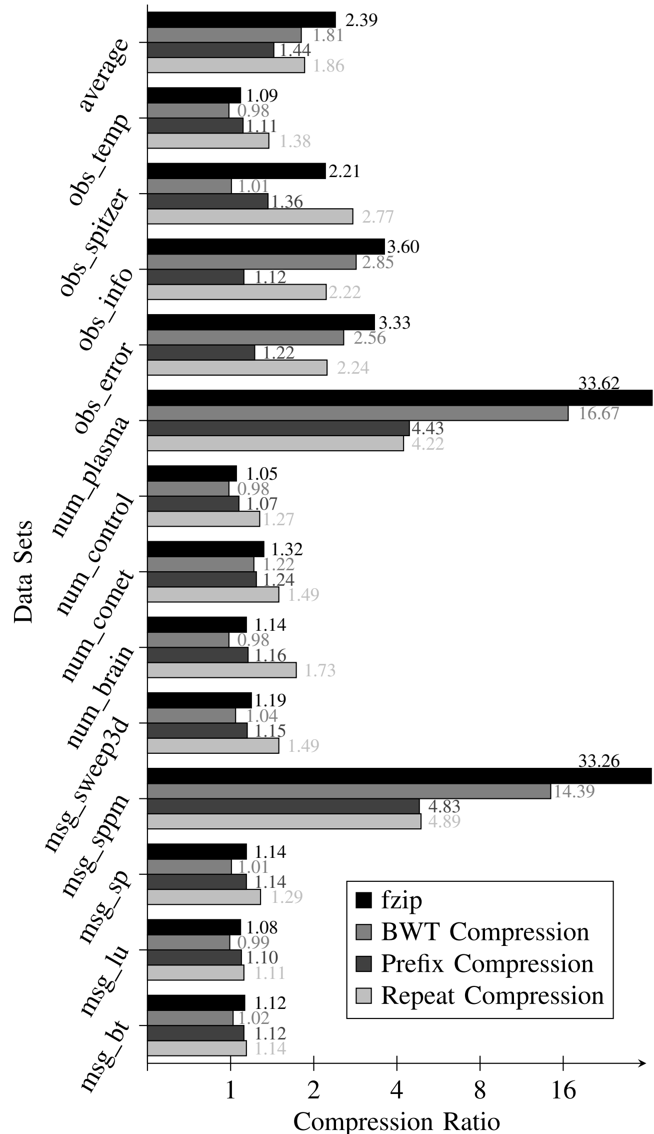



Fig. 6: This graph shows each of the three types of compression used to make up fzip. There are some cases where the overhead of combining the schemes outweighs the advantages.

### *B. Prefix Compression*

After BWT compression, there exists a new array of values. This array should no longer contain many long pattern sequences. However, patterns still exist among the values themselves. The values either repeat or partially repeat. Specifically, many values share common prefixes. fzip uses arithmetic codes to encode these common prefixes.

To begin with, fzip creates a large tree to represent all the values in the array. Fig. 7a shows an example tree for a small dataset. The tree follows the following rules: each node has up to two children. Each edge represents a 1 bit or a 0 bit. Each node in the tree represents a prefix. The root node represents "" or no prefix. Each node also has a weight, which represents the number of values with the prefix the node represents. So,

the weight of the root node equals the total number of values. The weight of the left (or 0 bit) child of the root represents the prefix "0". Its weight represents the number of values that start with "0" (all non-negative values). Likewise, the right child of the root represents the prefix "1" and its weight is the number of values starting with 1 (all the negative values).

Several properties appear. First, the sum of all the weights of the nodes in any level equals $n$, where $n$ is the total number of values. Moreover, the weight of any set of nodes that partitions the root node from the $65^{th}$ level (and does not contain more nodes than necessary to create the partition) equals $n$.

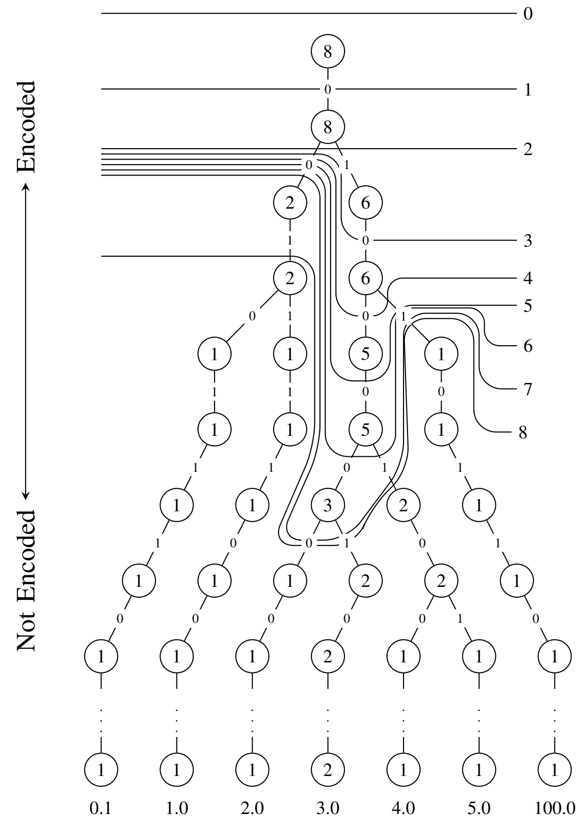
Second, the tree is unbalanced (in our case this is good). Put another way, the datasets contain an unequal number of positive and negative numbers, also any "normal" dataset would not have an exponential distribution from $2^{-12}$ to $2^{12}$ in such a way to make the rest of the tree balanced.

Tree creation starts with the root node, which has a starting weight of 0. To create the rest of the tree, add each value to the tree in the following way: Create a pointer to a "current node" $c$ and initiate $c$ to the root node. Increment the weight of $c$ (the root node). Then, with the most significant bit (the sign bit) of the floating point value, update $c$ by following the edge that matches this bit. If this edge does not exist create the edge and corresponding node. Then, increment the weight of the new $c$. This repeats until you reach the $64^{th}$ bit. Then, the next value gets added to the tree. This continues until the last value gets added to the tree.

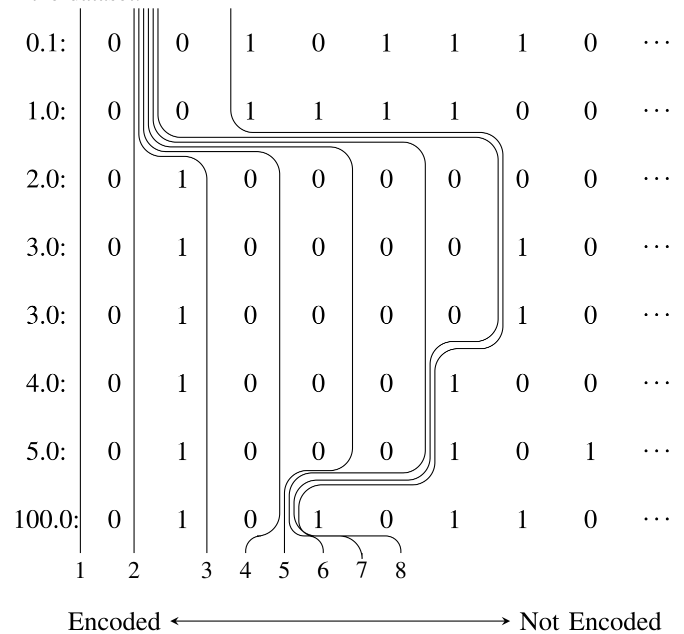
fzip calculates the prefix codes by creating a partition in the tree. To start, fzip creates a partition with only the root node. Then it includes the edge with the largest cut length in the partition. This repeats until a predetermined number of edges become cut by the partition. Using a list of prefix, prefix code tuples we can represent the encoding scheme of the first 8 partitions of the example in Fig. 7:

1) (0,)
2) (00,0), (01,1)
3) (00,0), (010,1)
4) (00,00), (0100,01), (0101,10)
5) (00,00), (01000,01), (0101,10)
6) (00,00), (010000,01), (010001,10), (0101,11)
7) (00,000), (0100000,001), (0100001,010), (010001,011), (0101,100)
8) (001,000), (0100000,001), (0100001,010), (010001,011), (0101,100)

Each added node improves the compression because of the following observation: Let the last added node equal $A$. The number of bits in the uncompressed (not-encoded) stream decreases by weight($A$). However, the code lengths have to increase because the partition cut-size ($k$) increases. The code lengths equal $\log_2(k)$. So the increase in the code length equals $\log_2(k+1) - \log_2(k)$ or $\frac{1}{k}$ by using derivatives. So the codes stream will increase by $\frac{n}{k}$, where $n$ equals to number of values in the data set. If you choose $A$ to maximize weight($A$) (a greedy algorithm) then weight($A$) $>$ average edge cut $> \frac{n}{k}$.




(a) Each node in the above tree represents every prefix that occurs in the dataset.




(b) The above sorted list of values gives a second visual representation of how the partition grows.

Fig. 7: The above 2 figures show the first 8 partition cuts in prefix compression for the example dataset {0.1, 1.0, 3.0, 5.0, 3.0, 100.0, 4.0, 2.0}. For simplicity, this example uses half-precision (16-bit) encoding.

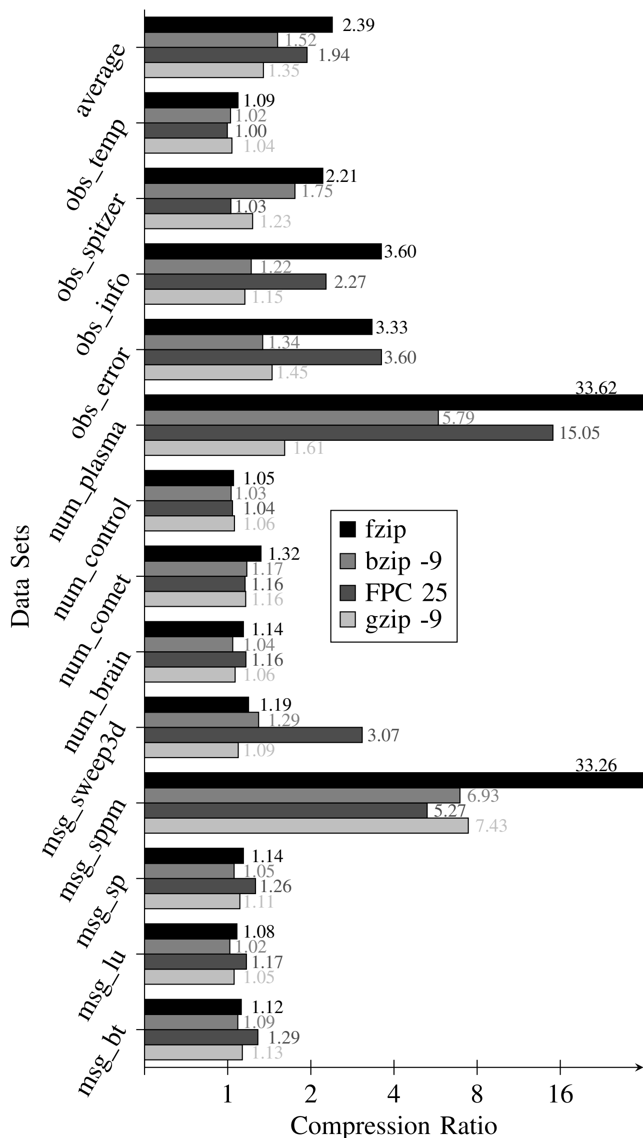



Fig. 8: The comparison of different compression schemes shows fzip performs well.

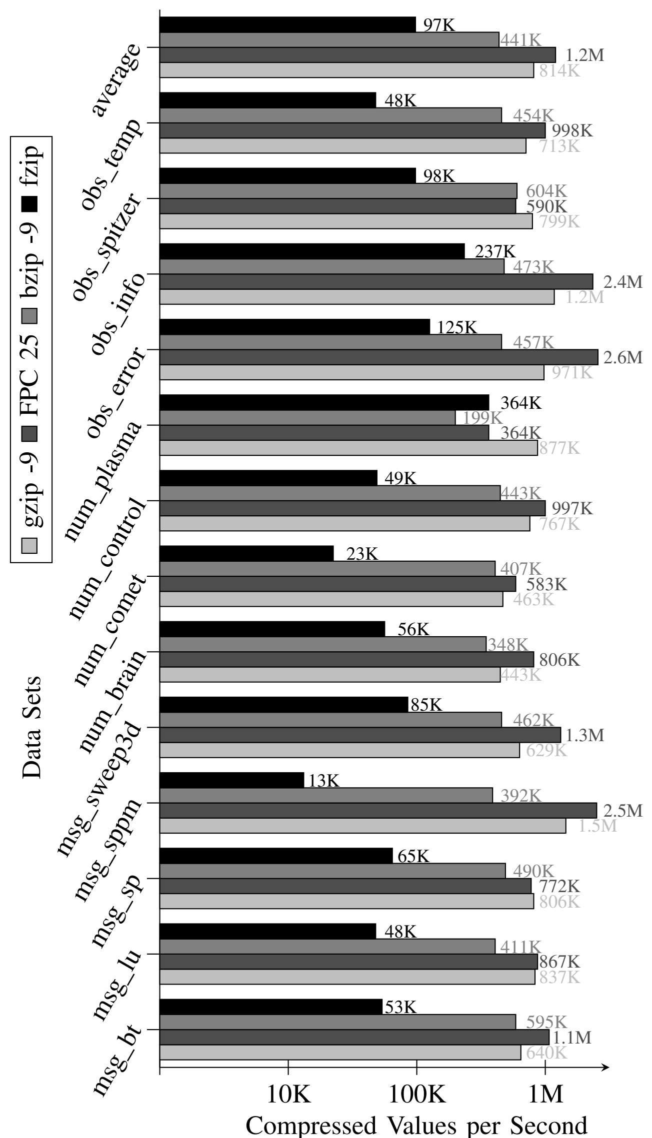



Fig. 9: The above compression runtime analysis shows that fzip has some improvement to make to compete with other programs's runtime.

Therefore, the total size of the prefix compression, excluding overhead, keeps improving as the partition increases.

But, what if a value occurs often? Say the value $1.0$ occurs $10\%$ of the time? Ideally you should encode $1.0$ as 4 bits ($\log_2 10$ rounded up), but if we continue to grow the partition beyond cutting 16 edges $1.0$ would encode as more than 4 bits. Our solution freezes the codes once a node from the last ($65^{th}$) level becomes included in the partition. This allows fzip to continue to improve prefix compression by growing the partition and also encode common values with shorter codes. This change makes the encoding to variable-length arithmetic encoding.

Of course, the overhead to store all of the codes exists. Currently, a 16 byte record describes each code. Each record stores the prefix, the prefix length and the code length. To balance the benefit of prefix encoding with its overhead, we limit the overhead to $1\%$ of the original array size.

### C. Repeated Value Extension

Prefix compression does not compress all of the repeated values. So, fzip extends prefix compression to specifically include all repeated values. Again explaining why repeated values compress well: All of the datasets have less than 37 million values. An index of 26 bits can address the entire dataset. Even if a value repeats only once (occurs twice) there still exists an advantage to store the repeated values in a repeated value array and store the indexes into this array instead of the original values. In the previous example $26 + 26 + 64 < 64 + 64$ (2 indices plus the value in the array equals less than storing 2 values).

To encode these repeats, we expanded the set of prefix codes. This increases the original code lengths by up to one bit. This seemed like a small trade off to make. All the repeats have the same length (64-bits) and the same code length so we can encode each as 8 bytes, instead of the 16 used for the prefixes.

## V. Results

When comparing fzip to other compressors we used the options that optimized for compression ratio (verses optimizing for compression speed). For FPC [5] we used option "25" and for gzip [7] and bzip2 [8] we used option "-9". We also tried to show the 3 algorithms used in fzip separately in Fig. 6. fzip performed the best with the 3 algorithms combined together. Occasionally, the overhead meant it did not perform better than one algorithm alone for some test cases. When compared to other programs (Fig. 8) fzip achieved the best average compression ratio.

However, we did not attempt to optimize runtime. So, as expected, fzip performs badly compared to other software (Fig. 9). In future work, we plan to improve runtime along with compression ratio.

## Acknowledgments

This work is supported in part by the National Science Foundation (NSF) under the awards CNS-1116810 and CCF-1149539.

## References

[1] J. Ke, M. Burtscher, and E. Speight, "Runtime compression of MPI messages to improve the performance and scalability of parallel applications," in *Proceedings of the IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2004, pp. 59–65.

[2] K. Kourtis, G. Goumas, and N. Koziris, "Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, Sep. 2008, pp. 511–519.

[3] K. Townsend and J. Zambreno, "Reduce, reuse, recycle ($R^3$): a design methodology for sparse matrix vector multiplication on reconfigurable platforms," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, Jun. 2013.

[4] D. Saloman and G. Motta, *Handbook of Data Compression*, 5th ed. London: Springer, 2010.

[5] M. Burtscher and P. Ratanaworabhan, "FPC: A high-speed compressor for double-precision floating-point data," *IEEE Transactions on Computers (TC)*, vol. 58, no. 1, pp. 18–31, Jan. 2009.

[6] M. Burrows and D. J. Wheeler, "A block-sorting lossess data compression algorithm," Systems Research Center, Technical Report 124, May 1994.

[7] (2013) The gzip home page. [Online]. Available: http://www.gzip.org/

[8] (2013) Official home page for bzip2. [Online]. Available: http://www.bzip.org/

[9] V. Engelson, D. Fritzson, and P. Fritzson, "Lossless compression of high-volume numerical data from simulations," in *Proceedings of the Data Compression Conference (DCC)*, Mar. 2000, pp. 574–586.

[10] B. Goeman, H. Vandierendonck, and K. Bosschere, "Differential FCM: Increasing value prediction accuracy by improving table usage efficiency," in *Proceedings of the IEEE International Symposium on High Performance Computing Architecture (HPCA)*, Jan. 2001, pp. 207–216.

[11] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 12, no. 5, pp. 1245–1250, Sep. 2006.

[12] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 1996, pp. 138–147.

[13] P. Ratanaworabhan, J. Ke, and M. Burtscher, "Fast lossless compression of scientific floating-point data," in *Proceedings of the Data Compression Conference (DCC)*, Mar. 2006, pp. 133–142.

[14] Y. Sazeides and J. E. Smith, "The predictability of data values," in *Proceedings of the ACM/IEEE International Symposium on Microarchitectures (MICRO)*, Dec. 1997, pp. 248–258.

[15] M. Schindler, "A fast renormalisation for arithmetic coding," in *Proceedings of the Data Compression Conference (DCC)*, Mar. 1998, p. 572.

[16] T. Richter, "Evaluation of floating point image compression," in *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, Nov. 2009, pp. 1909–1912.