gFPC: A Self-Tuning Compression Algorithm

Martin Burtscher
The University of Texas at Austin
burtscher@ices.utexas.edu

Paruj Ratanaworabhan
Kasetsart University
paruj.r@ku.ac.th

Abstract

This paper presents and evaluates gFPC, a self-tuning implementation of the FPC compression algorithm for double-precision floating-point data. gFPC uses a genetic algorithm to repeatedly reconfigure four hash-function parameters, which enables it to adapt to changes in the data during compression. Self tuning increases the harmonic-mean compression ratio on thirteen scientific datasets from 22% to 28% with sixteen kilobyte hash tables and from 36% to 43% with one megabyte hash tables. Individual datasets compress up to 1.72 times better. The self-tuning overhead reduces the compression speed by a factor of four but makes decompression faster because of the higher compression ratio. On a 2.93 GHz Xeon processor, gFPC compresses at a throughput of almost one gigabit per second and decompresses at over seven gigabits per second.

1. Introduction

Many compression algorithms are parameterizable. Some parameters, e.g., a table size, allow a straightforward trade-off between the compression ratio and speed. Such parameters are typically exported to the user as command-line options. Other parameters, e.g., a hash-function configuration, provide no obvious trade-off and interact in complex ways. The optimal setting of these parameters is often input dependent and may even change during compression. This paper presents an on-line, genetic-algorithm-based, self-tuning approach to automatically optimize such parameters and demonstrates how self tuning works on the example of the FPC compression algorithm [1].

In previous work, we have shown that FPC compresses our thirteen scientific datasets on average as well as other compression algorithms do, but one to two orders of magnitude faster [2]. However, compression speed is often less important than decompression speed and compression ratio, especially in environments where data are compressed once but decompressed many times. Our self-tuning approach increases the compression ratio at the cost of slower compression, but at no penalty in decompression speed, by repeatedly optimizing parameters during compression and recording their values in the compressed output to make them available to the decompressor. gFPC, a self-tuning implementation of the FPC algorithm, evolves four hash-function parameters in this way.

The results show that the empirically determined and hardcoded hash-function parameters in FPC are suboptimal even though 10,000 compression experiments were performed to optimize them [2]. In contrast, gFPC only requires the equivalent of four compression experiments and exceeds FPC's compression ratio by roughly five percent on average and by up to 72% on individual datasets. gFPC compresses better because it optimizes its hash function for each dataset and adapts to changes in the data on the fly. Due to the self-tuning overhead, gFPC compresses four times more slowly than FPC, but it decompresses faster since fewer bits are processed due to the higher compression ratio.

This paper is organized as follows. Section 2 provides an overview of gFPC and describes the self-tuning approach. Section 3 summarizes related work. Section 4 presents the evaluation methods. Section 5 discusses the results. Section 6 provides conclusions.

2. The gFPC algorithm

gFPC is based on FPC [1, 2] and 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* [3] and a *dfcm* [4] value predictor to predict the doubles. Both predictors are effectively hash tables, and we use the terms hash table and predictor interchangeably in this paper. The prediction 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. gFPC 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.

FPC outputs the compressed data in blocks. Each block has a header that specifies the configuration used, how many doubles the block encodes, and how long it is in bytes. The header is followed by the stream of four-bit codes and the stream of residual bytes.

Decompression of each double 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.

For performance reasons, gFPC interprets all doubles as 64-bit integers and uses only integer arithmetic. Additional information about the algorithm is available elsewhere [2].

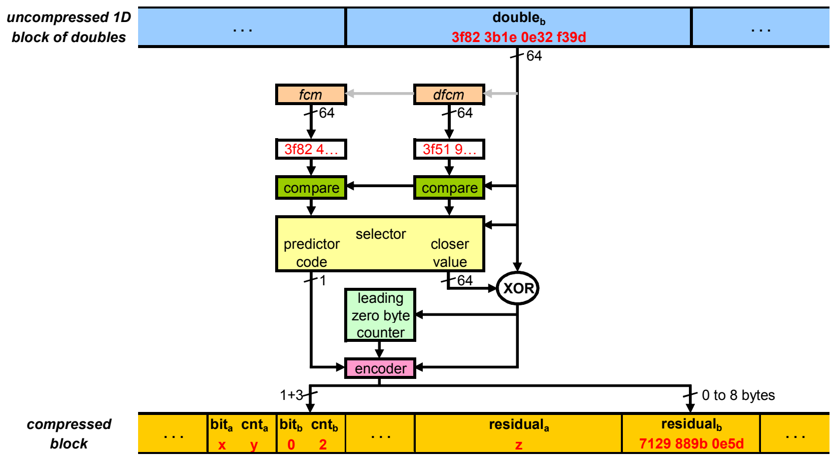

Figure 1: gFPC compression algorithm overview.

2.1 Predictor operation

Before compression and decompression, both predictor tables are initialized with zeros. After each prediction, they are updated with the true double value. 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 << hlshift) ^ (true_value >> hrshift)) & (table_size-1);
```

Right shifting `true_value` (i.e., the current double expressed as a 64-bit integer) by `hrshift` bits eliminates the often random least-significant mantissa bits. The remaining bits are xored with the previous hash to produce the new hash. However, the previous hash is first shifted by `hlshift` bits to the left to gradually phase out bits from older values [5]. Thus, the hash (`fcm_hash`) represents the sequence of the *k* most recently encountered doubles, and the hash table stores the doubles that follow these sequences. 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 differ. The complete C code is available at http://users.ices.utexas.edu/~burtscher/research/gFPC/.

```
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 << dlshift) ^ ((true_value - last_value) >> drshift))
  & (table_size - 1);
last_value = true_value;
```

The four shift amounts in the two hash functions are the target of the self tuning in gFPC. The two `_rshift` amounts can vary between zero and 63 and the two `_lshift` amounts can vary between one and $\log_2$(`table_size`). This yields 802,816 possible configurations with 256 kB of table space, making it too costly to try them all.

2.2 Genetic self tuning

Our genetic-algorithm-based self-tuning approach works by compressing each block of data multiple times with different hash-function configurations. The number of configurations tested is called population size. The initial population is seeded with the FPC configuration but is otherwise random. The fitness, i.e., the quality, of each configuration is the compression ratio achieved on the current block of data. The configuration with the highest fitness is written to the compressed output along with the compressed block of data it produced. The next block is compressed with a new generation of configurations. The smaller the block size the more often a new generation is produced.

Each configuration in the new generation is computed as follows. With a probability that is proportional to the fitness, two "parent" configurations from the previous generation are chosen. Their parameters are randomly combined by taking some of them from one parent and the remaining parameters from the other parent. The hope is that this "cross-over" operation will produce good combinations of parameters. Next, each resulting parameter is replaced with a probability of one third by a random value. This "mutation" operation should prevent the algorithm from getting stuck in a local maximum.

At the beginning of a new generation, the hash tables for each configuration must be reinitialized. For performance reasons, we opted not to zero out the tables but instead copy the table content from the best previous configuration into all tables of the new generation. This approach is almost as fast as zeroing out the tables, resulting in little speed degradation in the compressor, but yields better compression ratios because the tables are initialized with actually seen values and, more importantly, eliminates the need to reinitialize the hash tables in the decompressor, making decompression much faster.

3. Related work

Prior art that applies a genetic algorithm (GA) to data compression can be divided into two categories. In the first category, a genetic system evolves a program that outputs an approximation of the original uncompressed data. The system achieves compression if the evolved program's size is less than the size of the original data. Koza [6] was the first to apply this approach to data compression by lossily compressing a small bitmap. He coined the term programmatic compression to describe the approach. Nordin and Banzhaf [7] employed this scheme to compress image and sound data that are representative of real-world workloads. Krantz et al. [8] followed up on Nordin's work but targeted video data. These approaches are also lossy. Fukunaga and Stechert [9] used programmatic compression to evolve predictive coding compression algorithms to losslessly compress images. Salami et al. [10, 11, 12] developed a hardware approach to evolve similar predictive algorithms for image compression. The proposed hardware is capable of both lossy and lossless compression. The latter is achieved by removing the quantization process. Realizing that evolving a good compression program is both difficult and computationally expensive, Parent and Nowe [13] proposed to evolve a preprocessor instead. The output of the preprocessor is the input to a compressor. Decompression follows this two-stage process in reverse order.

Our work belongs to the second category, where a set of parameters related to the compression program is evolved instead of the whole compressor. The goal is to discover the optimal set of parameters to enhance the performance of the compressor. Klappenecker and May [14] improved upon conventional wavelet compression schemes by finding optimal parameters to minimize rate-distortion while maintaining compression ratios that are competitive with conventional schemes. Feiel and Ramakrishnan [15] employed a GA in the vector quantization process to optimize their color image compression system. Suman et al. [16] found a set of mappings, called fractal codes, through a GA for use in their fractal-based image compressor. Oroumchian et al. [17] developed a scheme to compress Unicode Persian text by constructing a dictionary using a GA to find the optimal set of *n*-grams for substitution. Intel patented a method [18] to compress microcode using a GA-based approach to find optimal representations of microcode in bit strings. Kattan and Poli [19] divided a file into smaller blocks and used a GA to find an optimal compressor for each block from a set of candidate compressors. All of this prior art uses GAs to optimize parameters off-line, which is a slow process. In contrast, we propose an on-line approach, which is not only much faster but can also result in higher compression ratios because the compressor is able to dynamically adapt to changes in the input data.

4. Evaluation methodology

4.1 System and compiler

We evaluated gFPC on an Intel Nehalem and an AMD Opteron system. The throughput results are somewhat lower on the Opteron, but the trends and relative performance are the same. Hence, we only present results for the Nehalem in this paper.

The Nehalem system is a Sun Fire X2270 64-bit Server running Ubuntu Linux version 8.06. It has 24 GB of memory and two quad-core 2.93 GHz Intel Xeon 5570 processors. We only use one thread on one core on an otherwise idle system. Each core has two 32 kB L1 caches and a unified 256 kB L2 cache. Each processor has an 8 MB L3 cache. We use the gcc compiler version 4.2.4 with the "-O3 -march=nocona -static -std=c99" flags.

4.2 Measurements

All timing measurements refer to the elapsed time reported by the UNIX shell command *time*. Each experiment was conducted four times in a row and the median running time of the last three runs is reported. This approach minimizes the overhead due to hard disk operations because, after the first run, the compressors' inputs are cached in main memory by the operating system. The output is written to /dev/null, i.e., it is consumed but ignored. All averages reported in this paper refer to the harmonic mean.

4.3 Datasets

We use thirteen real-world scientific datasets from multiple domains for our evaluation. Each dataset consists of a binary sequence of IEEE 754 double-precision floating-point values. Table 1 summarizes information about the datasets. 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, that is, 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, i.e., it reflects how close the first-order entropy is to that of a random dataset with the same number of unique values. More information about these datasets can be found elsewhere [2].

Table 1: Statistical information about the datasets.

		size (megabytes)	doubles (millions)	unique values (percent)	1st order entropy (bits)	randomness (percent)
MPI messages	bt	254.0	33.30	92.9	23.67	95.1
	lu	185.1	24.26	99.2	24.47	99.8
	sp	276.7	36.26	98.9	25.03	99.7
	sppm	266.1	34.87	10.2	11.24	51.6
	sweep3d	119.9	15.72	89.8	23.41	98.6
numeric simulations	brain	135.3	17.73	94.9	23.97	99.9
	comet	102.4	13.42	88.9	22.04	93.8
	control	152.1	19.94	98.5	24.14	99.6
	plasma	33.5	4.39	0.3	13.65	99.4
observation data	error	59.3	7.77	18.0	17.80	87.2
	info	18.1	2.37	23.9	18.07	94.5
	spitzer	189.0	24.77	5.7	17.36	85.0
	temp	38.1	4.99	100.0	22.25	100.0

5. Results

5.1 Population size

This section investigates how the population size affects the achievable compression ratio of gFPC. Because the compression speed is proportional to the population size, we want to identify the smallest size that yields good results. Figure 2 shows the harmonic-mean compression ratio over the thirteen datasets for a small, middle, and large predictor size and a population size of one through twenty. We chose 8 kilobytes, 256 kilobytes, and 8 megabytes for the predictor size because these sizes fit in many current systems' L1, L2, and L3 caches, respectively. Note that, with a population size of one, self tuning is disabled and gFPC defaults to FPC's configuration.

The results reveal that a population size between two and four is sufficient to reap almost the full benefit of self tuning for the three predictor sizes we tested. In all three cases, a population size of four yields a harmonic-mean compression ratio within half a percent of the maximum for that predictor size. Hence, we will use this population size in the rest of the paper.

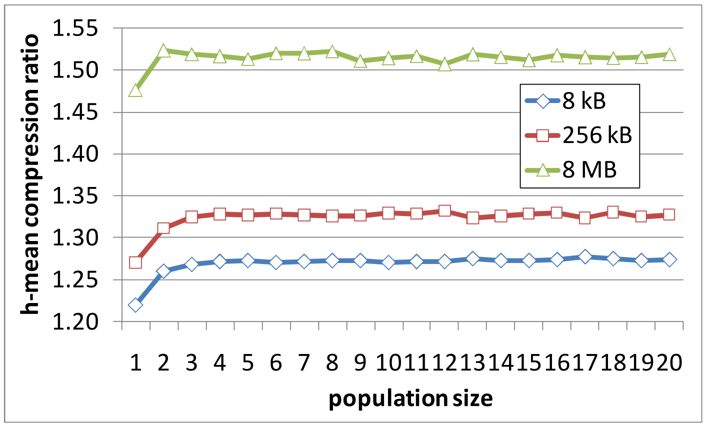

Figure 2: gFPC's harmonic-mean compression ratio for different population sizes.

5.2 Block size

This section studies the influence of the block size on the compression ratio of gFPC. The block size determines how many data values are compressed before the genetic algorithm updates the configuration. Larger block sizes reduce warm-up inefficiencies in the predictors but also reduce the number of reconfigurations. Because the former increases and the latter decreases the compression ratio, we expect medium-sized blocks to be a good trade-off. Figure 3 shows the harmonic-mean compression ratio over the thirteen datasets for the small, middle, and large predictor size and block sizes of four kilobytes through 256 megabytes in powers of two.

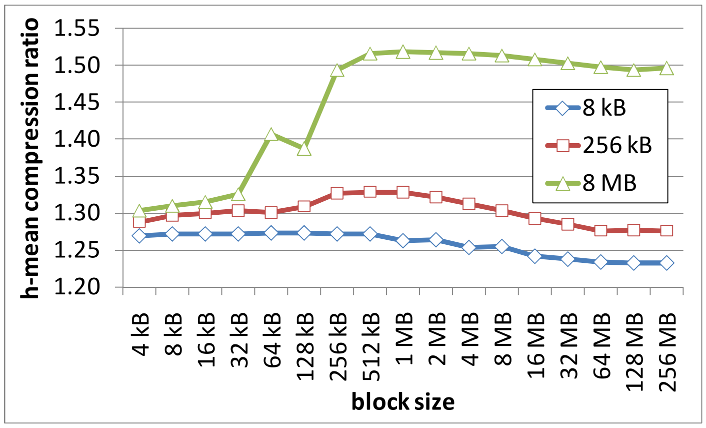

Figure 3: gFPC's harmonic-mean compression ratio for different block sizes.

The results confirm that the compression ratio indeed peaks for medium sized blocks. The peak is more pronounced for larger predictors, i.e., they depend more on the block size for good performance because of the aforementioned warm-up effect, which lasts longer with larger predictor sizes. The peak for the smallest predictor is at 64 kilobytes, for the middle predictor it is at 512 kilobytes, and for the largest predictor it is at one megabyte. However, all three predictors perform within 0.15% of their maximum for a block size of 512 kilobytes. Hence, we use this block size in the rest of the paper. Larger block sizes can hurt the performance by up to four percent in the range we investigated. Smaller block sizes are bad for large predictor sizes and hurt the compression ratio by up to 14%. Thus, it is more important to avoid block sizes below the optimum than block sizes above the optimum, especially for large predictors.

5.3 Compression ratio comparison

This section compares the compression ratio of gFPC with that of FPC, which utilizes a fixed hash-function configuration for all inputs and predictor sizes, and two enhanced versions of FPC. The first enhanced version, FPC_{size}, uses a different configuration for each predictor size. The second enhanced version, FPC_{all}, uses a different configuration for each predictor size and each input. The configurations of both enhanced versions were optimized using a conventional, off-line genetic algorithm combined with a local search.

The off-line genetic algorithm was run with a population size of sixteen and a random initial population until the compression ratio did not improve for twenty generations in a row, at which point the algorithm switched to the local search. The local search then exhaustively tried all 81 combinations of adding one, zero, and minus one to each of the four configuration parameters to determine whether the genetic algorithm had found a local maximum. If not, the best configuration identified by the local search was fed back to the genetic algorithm. This procedure repeated until a local maximum was found. We used this approach to determine the FPC_{size} and FPC_{all} configurations that maximize[1] the harmonic-mean compression ratio over the thirteen inputs. Figure 4 presents the results.

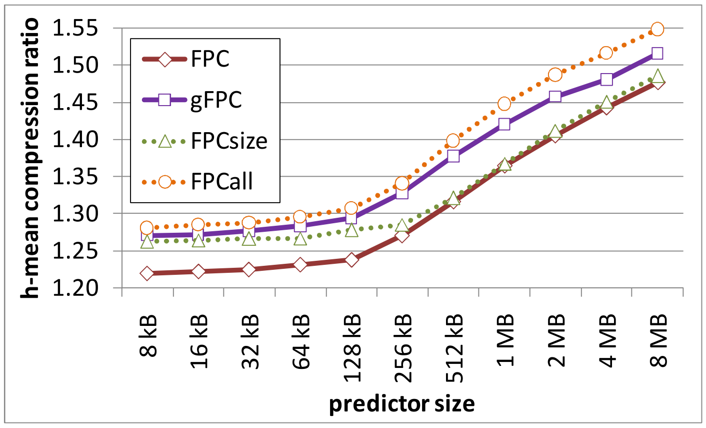

Figure 4: Harmonic-mean compression ratio for different predictor sizes of gFPC and FPC as well as two idealized versions of FPC.

gFPC outperforms FPC by four to five percent except on the two largest predictor sizes, where the compression ratio is 2.6% higher. These performance differences are substantial given that the datasets are only compressed by 22% to 51%.

Comparing FPC with FPC_{size}, we find the empirically obtained configuration for FPC to be close to optimal for 512-kilobyte and higher predictor sizes. For predictor sizes below 256 kilobytes, FPC misses about four percentage points in compression ratio.

gFPC outperforms FPC_{size}, especially for larger predictor sizes, because FPC_{size} uses the same configuration for all inputs whereas gFPC individually tunes itself for each in-

[1] The result is not guaranteed to be optimal but is assumed to be at least close.

put. gFPC does not reach the harmonic-mean compression ratio of FPC_{all}, though, because it uses earlier input data to generate configurations for compressing later data, which can result in suboptimal configurations. Nevertheless, it is possible for gFPC to exceed the performance of FPC_{all} because FPC_{all} uses the same configuration for the entire input whereas gFPC can change its configuration during compression. Accordingly, gFPC delivers up to 0.6% higher compression ratios than FPC_{all} on some of our datasets, but on average FPC_{all} compresses more. However, gFPC performs the equivalent of four dataset compressions per run whereas FPC_{all}'s off-line genetic algorithm and local search require an average of 1,267 compressions, meaning that gFPC is 317 times faster.

5.4 Self-tuning benefit

This section studies the increase in compression ratio due to self tuning on individual datasets, i.e., the difference between gFPC and FPC. The results are shown in Figure 5.

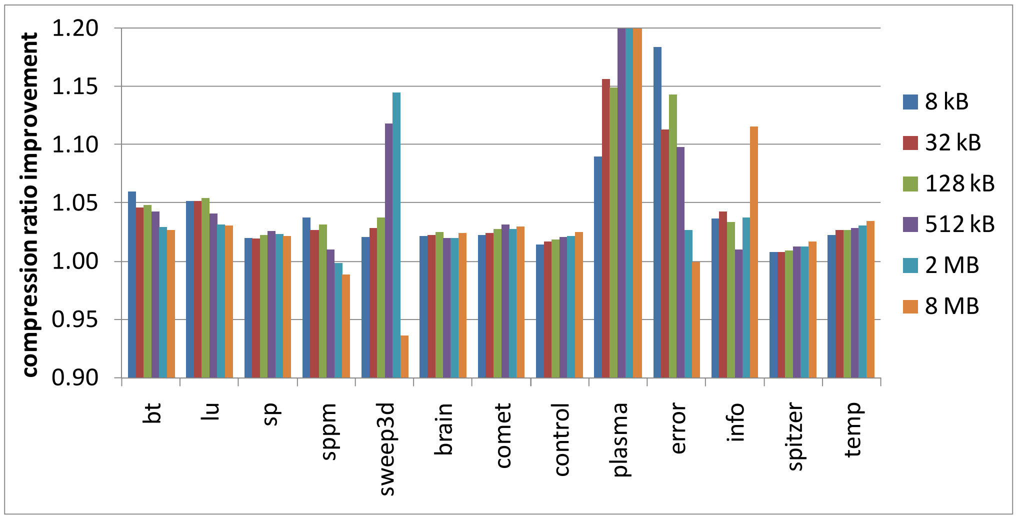

Figure 5: gFPC's improvement in compression ratio relative to FPC for each dataset. The *num_plasma* results for 512 kB, 2 MB, and 8 MB predictor sizes are 1.42, 1.72, and 1.28.

gFPC's tuned configurations perform between two and six percent better than FPC's for the majority of the datasets. Two datasets, *sppm* and *sweep3d*, perform worse on large predictors. Four datasets, *sweep3d*, *plasma*, *error*, and *info*, obtain a more than a ten percent improvement in compression ratio for at least one of the investigated predictor sizes. On *plasma*, gFPC delivers an up to 72% higher compression ratio than FPC does. These numbers demonstrate that self tuning can substantially boost the compression ratio. Note that FPC's configuration was optimized for these datasets. Hence, we expect the self-tuning benefit to be even greater on other datasets.

5.5 gFPC throughput

This section investigates the compression and decompression speed of gFPC with the standard population size of four, gFPC with a population size of one, and FPC. Figure 6 presents the results. With a population size of one, gFPC uses the same fixed hash-function configuration as FPC and does not perform any self tuning.

gFPC compresses about four times more slowly with a population size of four ($gFPC_4$) than with a population size of one ($gFPC_1$) due to the linear dependency between the compression speed and the population size. Decompression reuses the configuration that was selected during compression and is therefore not directly affected by the population size. Yet, $gFPC_4$ decompresses faster than both $gFPC_1$ and FPC do. The reason for the better performance is the higher compression ratio that $gFPC_4$ attains, which means that fewer data have to be processed during decompression, making it faster.

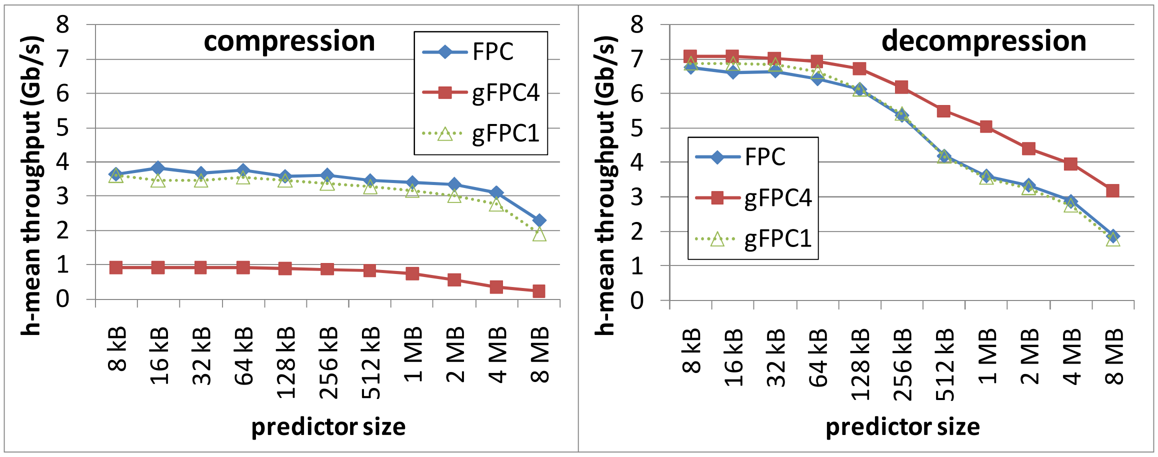

Figure 6: Harmonic-mean compression (left panel) and decompression (right panel) throughput of FPC and gFPC with a population size of one ($gFPC_1$) and four ($gFPC_4$).

$gFPC_1$ is faster than FPC at decompression for small predictor sizes and a little slower for large predictor sizes because gFPC is based on a new implementation that is optimized for x86 machines and small predictor sizes, which we believe to be the most common use case. In contrast, FPC's implementation is optimized for Itanium-based machines and larger predictor sizes. $gFPC_1$ is a little slower at compression than FPC because of the overhead due to the self-tuning machinery (which is useless for a population size of one but is still executed). Note that $gFPC_1$'s and FPC's compression ratios are identical because they both use the same hash-function parameters.

gFPC compresses the datasets at almost one gigabit per second while running the self-tuning algorithm with a population size of four and decompresses them at just over seven gigabits per second. The reported performance is for a single thread and can possibly be improved significantly with a parallel implementation, as we have done with FPC [20].

6. Summary and conclusions

This paper describes and evaluates gFPC, a self-tuning implementation of the FPC compression algorithm for double-precision floating-point data. The presented self-tuning approach is based on an on-line genetic algorithm that repeatedly optimizes and adapts four hash-function parameters to the data being compressed. The results show that self tuning generally improves the compression ratio, in some cases dramatically. The best block size, i.e., how quickly the algorithm should reconfigure the four parameters, correlates with the size of the hash table. A small population size is sufficient for good performance of the genetic algorithm, which is important because the compression speed decreases linearly with the population size. The decompression speed is not directly affected. gFPC compresses data at a throughput of almost one gigabit per second on a 2.93 GHz Xeon processor, demonstrating that our self-tuning algorithm can operate at a high speed.

We believe the presented approach is general and applies to other parameterizable compression algorithms as well. The C source code of gFPC, which is freely available at http://users.ices.utexas.edu/~burtscher/research/gFPC/, can easily be adapted to accommodate other compressors by replacing the two functions that compress and decompress a block of data. Tuning fewer or more than four parameters should also be straightforward. Hence, we think that our technique as well as the gFPC code base represent a general framework for making compression algorithms self tuning, thus allowing them to trade off a lower compression speed for a higher compression ratio.

7. References

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

[2] M. Burtscher and P. Ratanaworabhan. FPC: A High-Speed Compressor for Double-Precision Floating-Point Data. IEEE Transactions on Computers, Vol. 58, No. 1, pp. 18-31. 2009.

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

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

[5] Y. Sazeides and J. E. Smith. "Implementations of Context Based Value Predictors." Technical Report ECE-97-8, University of Wisconsin-Madison. December 1997.

[6] J. R. Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press. 1992.

[7] P. Nordin and W. Banzhaf. Programmatic Compression of Images and Sound. First Annual Conference on Genetic Programming, pp. 345-350. 1996.

[8] T. Krantz, O. Lindberg, G. Thorburn, and P. Nordin. Programmatic Compression of Natural Video. Late Breaking Papers at the Genetic and Evolutionary Computation Conference, pp. 301-307. 2002.

[9] A. Fukunaga and A. Stechert. Evolving Nonlinear Predictive Models for Lossless Image Compression with Genetic Programming. Third Annual Conference on Genetic Programming, pp. 95-102. 1998.

[10] M. Salami, M. Murakawa, and T. Higuchi. Data Compression based on Evolvable Hardware. International Conference on Evolvable Systems. 1996.

[11] M. Salami, M. Murakawa, and T. Higuchi. Lossy Image Compression by Evolvable Hardware. Evolvable Systems Workshop. 1997.

[12] M. Salami, M. Iwata, and T. Higuchi. Lossless Image Compression by Evolvable Hardware. European Conference on Artificial Life. 1997.

[13] J. Parent and A. Nowe. Evolving Compression Preprocessors with Genetic Programming. Genetic and Evolutionary Computation Conference, pp. 861-867. 2002.

[14] A. Klappenecker and F. U. May. Evolving Better Wavelet Compression Schemes. Wavelet Applications in Signal and Image Processing III, Vol. 2569. 1995.

[15] H. Feiel and S. Ramakrishnan. A Genetic Approach to Color Image Compression. Symposium on Applied Computing. 1997.

[16] S. K. Mitra, C. A. Murthy, and M. K. Kundu. A Technique for Fractal Image Compression using a Genetic Algorithm. IEEE Transactions on Image Processing, Vol. 7, No. 4. 1998.

[17] F. Oroumchian, E. Darrudi, F. Taghiyareh, and N. Angoshtari. Experiments with Persian Text Compression for Web. 13th International World Wide Web Conference, pp. 478-479. 2004.

[18] Y. Wu and M. Breternitz. Genetic Algorithm for Microcode Compression. US Patent No. 7,451,121 B2. 2008.

[19] A. Kattan and R. Poli. Evolutionary Lossless Compression with GP-ZIP. 10th Annual Conference on Genetic and Evolutionary Computation, pp. 1211-1218. 2008.

[20] M. Burtscher and P. Ratanaworabhan. pFPC: A Parallel Compressor for Floating-Point Data. Data Compression Conference, pp. 43-52. March 2009.