Real-Time Synthesis of Compression Algorithms for Scientific Data

Martin Burtscher, Hari Mukka, Annie Yang, and Farbod Hesaaraki
Department of Computer Science
Texas State University
San Marcos, TX, USA

***Abstract*—Many scientific programs produce large amounts of floating-point data that are saved for later use. To minimize the storage requirement, it is worthwhile to compress such data as much as possible. However, existing algorithms tend to compress floating-point data relatively poorly. As a remedy, we have developed FPcrush, a tool that automatically synthesizes an optimized compressor for each given input. The synthesized algorithms are lossless and parallelized using OpenMP. This paper describes how FPcrush is able to perform this synthesis in real-time, i.e., even when accounting for the synthesis overhead, it compresses the 16 tested real-world single- and double-precision data files more quickly than parallel bzip2. Decompression is an order of magnitude faster and exceeds the throughput of multi-core implementations of bzip2, gzip, and FPC. On all but two of the tested files, as well as on average, the customized algorithms deliver higher compression ratios than the other three tools.**

Keywords—data compression; real-time algorithm synthesis

I. Introduction

High-performance computing systems produce scientific data at a rapidly growing rate. Many of the resulting files are made available for download or are archived for later use. In either case, data compression has the potential to reduce the required storage space and to boost the download speed. However, scientific data consist predominantly of floating-point values, which are often difficult to compress well [9].

Due to the wide availability and generality of gzip and bzip2, many HPC users compress their data with one of these tools, either directly or through the use of a corresponding compression "filter" in HDF5. FPcrush, the approach described in this paper, is meant as an alternative to using bzip2, gzip, or parallel versions thereof. In particular, FPcrush is a standalone tool for the lossless compression/decompression of files that is specifically tuned to work well on hard-to-compress floating-point data (but also works on other files).

In environments where data files are accessed often, fast decompression is important, preferably at the same or a higher speed than obtained when reading the uncompressed data. Fast compression is also desirable but conflicts with the goal of achieving high compression ratios because better compression generally requires more computation. Therefore, an asymmetric approach where compression takes substantially longer than decompression is probably unavoidable when aiming both for maximal compression and for fast decompression, which is the objective of our work.

Since data differ from each other and we want to compress each file as much as possible, it is unlikely that a single algorithm will suffice. This is why we propose customization, that is, to compress each input with a different algorithm. Unfortunately, only a handful of compression algorithms for floating-point data exist in the current literature. Rather than designing a few more algorithms, we created a framework that is capable of synthesizing millions of data compression algorithms [26]. From it, we derived FPcrush, which incorporates key techniques to quickly identify promising algorithms in its search space for compressing a given file. The resulting compressed files include a few bytes of information to identify the needed decompression algorithm.

At a high level, most data compression tools operate on a linear sequence of values stored in an array and comprise two main steps, a data model and a coder. Roughly speaking, the goal of the model is to accurately predict the input values. The residual (i.e., the difference) between each actual value and its predicted value will be close to zero if the model is accurate for the given data. This residual sequence of values is then compressed with the coder by mapping the residuals in such a way that frequently encountered values or patterns produce a shorter output than infrequently encountered data. The inverse operations are performed to decompress the data. For instance, an inverse model takes a residual sequence as input and generates the original sequence of values as output.

To be able to search for effective floating-point compression algorithms, we built a framework for synthesizing compressors and the corresponding decompressors using the following approach. We started with an in-depth study of previously proposed floating-point compression algorithms, broke them down into their constituent parts, rejected all parts that could not be implemented to run in linear time, and generalized the remaining parts as much as possible. This yielded a number of *algorithmic components* for building data models and coders. We then implemented each component using a common interface, i.e., each component can be given a block of data as input, which it transforms into an output block of data. This design makes it possible to *chain* the components, allowing the framework to generate a large number of compression-algorithm candidates from a set of components. Note

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00 ©2016 IEEE

that each component comes with an inverse that performs the opposite transformation. Thus, for any chain of components, which represents a compression algorithm, it is always possible to synthesize the matching decompressor. Figure 1 illustrates this approach on the example of three components named LVs, BIT, and LZ3 as well as a Cut. Section 2 explains the operation of these components. Note that the order of the components matters. Every permutation represents a distinct algorithm that yields a potentially different compression ratio.

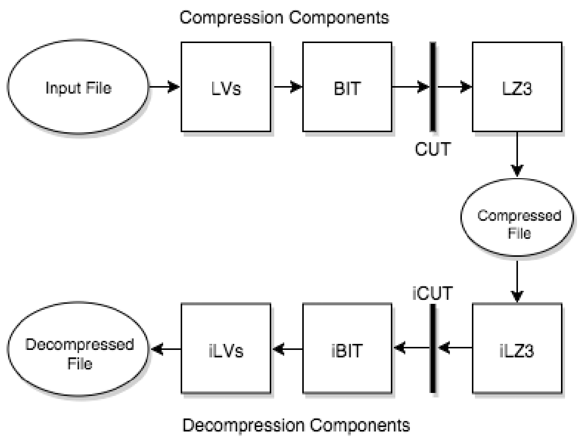

Fig. 1. Three chained components plus a Cut that represent a compression algorithm along with the corresponding inverse components that make up the decompression algorithm

Since we consider 29 algorithmic components, performing an exhaustive search to identify the best algorithm is too expensive for chains with more than about three components, i.e., algorithms with more than three stages. To speed up the search, FPcrush employs two complementary techniques. First, it uses a genetic algorithm (GA) to quickly find effective compressors without visiting most of the search space. Second, it extracts a short but representative segment from the given input and uses only this segment to drive the GA. The combination of these two approaches quickly yields a good algorithm that is then used to compress the entire input.

To further accelerate FPcrush, the synthesis as well as the resulting compression and decompression algorithms are parallelized using OpenMP. In particular, the GA evaluates the individuals in its population (cf. Section 3.2) concurrently. Moreover, the synthesized algorithms divide the input into chunks and process them in parallel. Since the evaluated individuals and the processed chunks are independent, these computations are embarrassingly parallel. As a consequence, their parallelization is trivial and not the focus of this paper.

We implemented all algorithmic components using C++ templates, which makes it possible to support both single- and double-precision data. Note that FPcrush uses a 4- and 8-byte *integer* representation of the floating-point data (i.e., the bit pattern representing the floating-point value is copied verbatim into an appropriately sized integer variable) and exclusively uses integer operations to maximize performance and to avoid the possibility of floating-point exceptions or rounding inaccuracies. In other words, FPcrush generates lossless integer compression algorithms that are fed with and tuned for floating-point data. This approach is also used by other floating-point compressors [3, 20]. As a result, positive and negative zeros and infinities, NaNs, denormals, and all other values are recreated bit-by-bit during decompression.

FPcrush's goal is to maximize the compression ratio on floating-point data while still providing high throughput, especially decompression throughput. This paper makes the following main contributions.

- It is the first work to demonstrate that superior compression algorithms can be synthesized in real-time and to present a combination of techniques that make it possible to achieve such high synthesis speeds.
- It shows that customized algorithms can yield higher compression ratios than pre-existing algorithms, both on single- and double-precision floating-point data.
- It illustrates that the synthesized parallel algorithms compress faster than parallel bzip2 and decompress more quickly than parallel gzip, bzip2, and FPC.
- It makes FPcrush freely available on the computers listed at cs.txstate.edu/~burtscher/research/FPcrush/.

The rest of this paper is organized as follows. Section 2 describes the algorithmic components used in FPcrush. Section 3 explains their implementation. Section 4 summarizes related work. Section 5 provides an overview of the evaluation system and the input files. Section 6 studies and analyzes various aspects of FPcrush. Section 7 concludes with a summary.

II. Algorithmic Components

This section describes the 29 algorithmic components available to FPcrush for synthesizing compression algorithms. Many of them are generalizations or approximations of components extracted from previously proposed algorithms. Each component takes a sequence of values as input (i.e., an array), transforms it, and outputs the transformed sequence. To organize the components, we grouped them into categories.

A. Mutators

Mutators are components that computationally transform each value in the sequence. This is done independently of any other value and does not change the length of the sequence.

The **NUL** component performs the identity operation, that is, it simply outputs the input sequence. Its presence ensures that chains with *n* components can also represent all possible algorithms with fewer than *n* components. NUL has the highest priority, i.e., FPcrush gives preference to shorter chains over longer chains with the same compression ratio.

The **SMS** component converts each value from sign-magnitude (as used in the IEEE 754 floating-point format) into signed twos-complement representation. It does this by inverting all but the most significant bit if the most significant bit is set. Since FPcrush processes all values using integer operations, this transformation may be beneficial.

B. Shufflers

Shufflers rearrange the order of the values in the sequence but perform no computation on them. Some shufflers reorder the bits or bytes within the values. None of them change the length of the sequence. In some cases, they operate on chunks of data that encompass multiple words.

The **BIT** component groups the values into chunks of as many values as there are bits per value. It then transforms each chunk independently by creating and emitting a word that contains the most significant bits of the values, followed by a word that contains the second most significant bits, etc. The resulting sequence is easier to compress in cases where the bit positions between consecutive input values exhibit a higher correlation than the values themselves.

The **ROT*n*** component takes a parameter n that specifies by how many units to rotate the bits of each word in the input sequence. There are seven versions of this component. For double-precision data, the values can be rotated by one to seven bytes, for single-precision data by one to seven nibbles, and for byte-sized data by one to seven bits. This rotation affects the behavior of some of the other components.

The **DIM*n*** component takes a parameter n that specifies the dimensionality of the input sequence and groups the values accordingly. For example, a dimension of three changes the linear sequence x_1, y_1, z_1, x_2, y_2, z_2, x_3, y_3, z_3 into x_1, x_2, x_3, y_1, y_2, y_3, z_1, z_2, z_3. This may be beneficial as values belonging to the same dimension often correlate more with each other than with values from other dimensions. For single-precision inputs, we use $n = 2, 3, 4, 5, 7, 8, 12$, and 32. For double-precision inputs, we use $n = 2, 3, 5, 7, 8, 12$, and 64. Since a dimensionality of $k \cdot m$ can be represented by combining a DIMk with a DIMm component, we primarily use small prime numbers for the parameter n. To capture important non-prime values of n in a single component (for performance reasons), we also include the following: $n = 12$ is useful because twelve is the least common multiple of 2, 3, 4, and 6 and therefore works well on 2D, 3D, 4D, 6D, and 12D data. $n = 8$ is useful in combination with the BIT component because there are eight bits per byte and, in case of double-precision values, there are eight bytes per word. The bytes per word is also the reason for including $n = 4$ in case of single-precision data. The largest included dimension reflects the number of bits per word, i.e., $n = 32$ for single precision and $n = 64$ for double precision.

C. Predictors

Predictors guess the current value based on previous values in the input sequence, subtract the predicted from the current value, and emit the result of the subtraction, that is, the residual sequence. If the predictions are close to the actual values, the residuals will cluster around zero, making them easier to compress than the original sequence. Predictors do not change the length of the sequence. The subtraction to compute the residual can be performed at word granularity (using conventional subtraction, denoted by a trailing '**s**') or at bit granularity (using XOR, denoted by a trailing '**x**').

The **LVs** and **LVx** components use the previous value in the sequence as a prediction of the current value. This is commonly referred to as delta modulation.

For n-dimensional data, it may be useful to utilize the n^{th} prior value. However, this can be achieved by preceding the LV component by a DIMn component. Hence, we do not include a last-n value predictor component.

D. Reducers

Reducers are the only components that can change the length of the sequence and therefore compress it. They exploit various types of redundancies to do so. The last component of a chain must always be a reducer and there has to be at least one reducer in each chain to form a useful compression algorithm. FPcrush enforces this restriction automatically and does not consider other chains.

The **ZE** component emits a bitmap that contains one bit for each value in the input. Each bit indicates whether the corresponding value is a zero or not. Following the bitmap, ZE emits all non-zero values from the input sequence. This component's effectiveness depends on the number of zeros, which is why some of the previously described components aim at generating as many zeros as possible.

The **RLE** component performs run-length encoding. In particular, it counts how many times the current value appears in a row. Then it counts how many non-repeating values follow. Both counts are recorded in a single word, i.e., each count gets half of the bits. This "count" is emitted first, followed by the current value and finally the non-repeating values. This procedure repeats until the end of the input is reached. We selected this specific version of run-length encoding because it proved particularly effective on hard-to-compress floating-point data. Nevertheless, its effectiveness is contingent upon the input containing many repeating values.

The **LZ*n*** component implements a variant of the LZ77 algorithm [27]. It incorporates tradeoffs that make it more efficient than other LZ77 versions on floating-point data and operates as follows. It uses a hash table to identify the location l of the most recent prior occurrence of the current value. Then it checks whether the n values immediately before location l match the n values just before the current location. If they do not, the current value is emitted and the component advances to the next value. If the n values match, the component counts how many values following the current value match the values following location l. The length of the matching substring is emitted and the component advances by that many values. Smaller values of n yield more matches, which have the potential to improve compression, but also result in a higher chance of zero-length substrings, which hurt compression. We consider $n = 1, 2, 3, 4, 5, 6$, and 7.

The LZ components contain a hash table. We use a constant table size of 65,536 words throughout this paper, which seems to work well and is reasonably fast. Note that larger tables tend to increase the compression ratio while decreasing the compression and decompression throughput. Smaller tables have the opposite effect.

E. The Cut

The **|** pseudo component, called the Cut and denoted by a vertical bar, is a singleton component that converts a sequence of words into a sequence of bytes. It is merely a type cast and requires no computation or data copying. Every algorithm produced by FPcrush contains exactly one Cut, which is included because it is often more effective to perform compression at byte rather than word granularity. Note, however, that the Cut can appear before the first component, in which case the data are treated as a sequence of bytes, after the last component, in which case the data are treated as a sequence of words, or between components, in which case the data are initially treated as words and then as bytes.

F. Discussion

The components described above can all be implemented to run in O(n) time, where n is the sequence length. We excluded more complex components such as move-to-front and block-sorting components to make the synthesis, which has to evaluate many different chains of components, as fast as possible. Nevertheless, as the results in this paper demonstrate, the included components suffice to create algorithms that compress better than preexisting tools in many cases.

Due to the Cut, FPcrush needs three versions of each component: one for double-precision values (8-byte words), one for single-precision values (4-byte words), and one for byte values. We implemented all components in form of C++ templates to facilitate the generation of these different versions.

Each component requires a corresponding inverse component that performs the reverse transformation. By chaining the inverse components in the opposite direction, FPcrush can automatically synthesize the matching decompression algorithm for any given chain of components, i.e., for any of the compression algorithms in can generate (cf. Figure 1).

The seven LZ components utilize hash tables. For performance reasons, their hash functions only use some of the bits from the input values. This is why altering the location of bits and bytes by other components affects the effectiveness of these components. Note, however, that FPcrush is able to optimize which bits to use by the hash function, for example, through the inclusion of an appropriate ROT component.

Not counting the Cut, FPcrush has 29 components at its disposal, only nine of which are able to reduce the length of the data. The purpose of the remaining 20 components is to transform the values in such a way that the reducers become maximally efficient. Thus, longer chains of components have the potential to compress better but make the search for a good algorithm take longer and decompression slower. For an algorithm with k stages, that is, a chain with k components, the search space encompasses $(k+1) \cdot 29^{k-1} \cdot 9$ possible algorithms because there are k+1 locations for the Cut, k–1 stages that can each hold any one of the 29 components, and a final stage that can hold any one of the nine reducers. This amounts to over a million possible four-stage algorithms and over 42 billion seven-stage algorithms.

III. FPcrush Implementation

A. Representative Segments

Since there is not nearly enough time available to search the entire input for a good compression algorithm, FPcrush first determines a small representative segment and then performs the search only on this segment. To identify a good segment, it breaks up the input into many segments and uses a sliding-window approach where the window is advanced in steps of one eighths of the segment size. The segment size is selectable and expressed as a percentage of the entire file size. Then, FPcrush computes the byte entropy of each segment and selects the one whose entropy is closest to that of the entire file. Shorter segments result in faster searches but may be less representative. Moreover, there is no guarantee that a representative segment exists at all. Once a good compression algorithm for the selected segment has been found, this algorithm is applied to the entire file.

B. Genetic Algorithm

The genetic algorithm (GA) [10, 11] employed by FPcrush to quickly search for an effective compression algorithm uses a fixed population size of twenty individuals. Each individual (i.e., a chain of components representing a compression algorithm) is initialized with a randomly selected reducer in the last stage, randomly selected components in the other stages, and a random location for the Cut. This population is then evolved over a selectable number of generations in the following way. First, the compression ratio (i.e., the fitness) of each individual is evaluated on the chosen segment and the best-performing algorithm is recorded. Then a new generation of individuals is created using the following genetic approach.

A quarter of the new individuals are the result of a cross-over operation, which selects two parents from the prior generation with a probability that is proportional to their fitness. The components are taken from one parent up to a randomly selected stage and the remaining components are taken from the other parent. The Cut is randomly taken from the first or the second parent. The next quarter of the new individuals are also the result of a cross-over operation. However, this cross-over picks the Cut and the components for each stage from one or the other parent based on a random bitmask. The third quarter of the new individuals are the result of mutating a single clone, which is selected from the prior generation with a probability proportional to its fitness. The mutation replaces one randomly selected component or the Cut with a random but legal alternative. The probability of a single mutation is 100%, a second mutation happens with 50% probability, a third one with 25%, etc. The last quarter of the new individuals are also the result of mutations, but in this case they are applied to a copy of the best algorithm found so far.

Genetic algorithms represent a heuristic search method that is meant to quickly converge on some good solutions. However, there is no guarantee that a genetic algorithm will do so and it generally does not find the globally best solution in large search spaces. However, the best identified solution often performs nearly as well as the globally best solution.

IV. Related Work

A. Floating-Point Compressors

This subsection summarizes related work on lossless floating-point compression. We extracted the basic idea behind many of our algorithmic components from these papers. Of course, many more papers on the lossless compression of floating-point data exist (cf. [19] and references therein).

Lindstrom and Isenburg discuss on-line compression of floating-point grid data for speeding up I/O operations [20]. They use a Lorenzo predictor and map reals to unsigned integers. FPcrush also exclusively uses integer representation and operations. Since the Lorenzo predictor is not particularly suitable for linear sequences of values, FPcrush does not include a corresponding component.

Burtscher and Ratanaworabhan's FPC algorithm targets double-precision values [3]. It predicts the integer interpretation of the 64-bit values using an FCM and a DFCM predictor. The two predictions are XORed with the true value. The result with more leading zeros is compressed using leading-zero byte counts. The authors also published a parallel version of their compression algorithm, called pFPC [4], with which we compare FPcrush in the result section. We include the XOR idea in our study. We found the FCM and DFCM predictors with leading-zero-byte-elimination to be outperformed by chains with an LZ component, which is why we ended up not including components for the two predictors.

Chen et al.'s work orders grid points of tetrahedral volume data to improve compressibility [6]. Their approach separates the "signed exponent" from the mantissa values. We include a similar component (BIT) that groups the various bit positions from adjacent values so that all the sign bits, exponent bits, etc. can be compressed together.

Bicer et al. describe a framework that XORs values and leading-zero compresses the results [1]. As it operates at bit granularity, their approach works for both single- and double-precision data. The data are split into chunks, which are compressed independently. FPcrush also supports both single- and double-precision data and uses data chunks to facilitate parallel compression and decompression.

Filgueira et al. focus on runtime compression of MPI messages, including floating-point messages [8]. They found lzop to work best on their synthetic integer and floating-point data that include a significant number of zeros because lzop is very fast. The user can select which compression algorithm to use for which data type. A later paper describes an extension that dynamically selects the most appropriate algorithm based on the data type, including none for short messages [9]. Our approach is orthogonal to theirs and could be used to find good compression algorithms for various data types.

Schendel et al. introduce a pre-compression tool to improve the performance of general-purpose compressors on double-precision floating-point data. Their approach analyzes the compressibility of the data at byte granularity, determines the best compressor for the job, and identifies and removes hard-to-compress sections before piping the remaining data to the compressor [22]. FPcrush searches for effective algorithms at word and byte granularity and produces customized, standalone compression algorithms. However, it uses a similar analysis of the input to find a good segment.

Jenkins et al. create a system for rapid indexing, storing, and querying based on compressed metadata [17]. Their compression approach is based on the idea that most double-precision data have similarity in the sign and exponent fields. They discard the redundancies in the higher-order bits and map the lower-order bytes to a bin according to distinct higher-order bits. They then pass the separated data to bzip2. FPcrush does not depend on such assumptions about the data, but it does contain components to separate higher- and lower-order bits so that they can be compressed separately.

B. Generating Compression Algorithms

This subsection describes prior techniques for synthesizing compression algorithms. FPcrush is a derivative of the Crusher framework, which we previously used to synthesize a floating-point compression algorithm that is GPU friendly [26]. This prior work uses some of the same algorithmic components to generate the compressor. However, it only employs components that can easily be parallelized for GPUs. FPcrush does not have this limitation, which is why its algorithms almost always compress the same files better, in some cases by a large margin. More importantly, our prior work is not concerned with the synthesis speed. It does not use a genetic algorithm nor segments to accelerate the processing. In fact, it is several orders of magnitude slower and unfit for real-time synthesis. Furthermore, it only proposes a single algorithm. In contrast, FPcrush generates a new algorithm for each file.

None of the remaining related works described in this subsection are designed for floating-point data. Instead, they target program execution traces, heterogeneous files, images, and databases. Moreover, none of these approaches were designed for or support real-time compression and none of them employ segmentation to speed up the algorithm generation. As a consequence, when including the synthesis time, they are much slower than standard compression tools. Hence, we do not compare FPcrush to these approaches in the result section.

Burtscher and Sam present TCgen, a tool that generates customized trace compressors based on a user-provided configuration of one or more predictors [5]. TCgen then translates this description into C source code that is optimized for the specified trace format and predictors. FPcrush supports a larger number of components, in particular also non-predictor components, and automatically determines good algorithms without the need for a description from the user.

Kattan and Poli propose a system that employs genetic programming to find optimal ways to combine standard compression algorithms [18]. They group similar data chunks together and label each group with the best compression algorithm for its chunks. We also utilize a genetic algorithm and combine components. However, their components represent entire compression algorithms whereas our components are finer grained and represent parts of a compression algorithm.

Hsu and Zwarico present an automatic synthesis technique for compressing heterogeneous files [12]. Each chunk of data is compressed using a different algorithm, which is determined using a statistical method. A compression history, required for decompression, is automatically generated and added in this phase. We use a similar approach to record the needed decompression algorithm in the compressed output.

Mitra et al. propose a methodology for compressing fractal images using a genetic algorithm [21]. Initially, fractal codes are computed for each domain block. Then these blocks are classified into two types based on the variability of the pixels in each block. A block belongs to the smooth type if its variance is below a given threshold and is considered rough otherwise. The purpose of this classification is to obtain higher compression ratios and to reduce the encoding time. The final step uses a genetic algorithm to find a good match for the rough domain blocks. Wu and Lin use a similar approach with three classes [25]. FPcrush also uses a genetic algorithm to find an effective solution in its search space.

Several other papers have been published that employ a genetic algorithm for image compression, primarily to speed up the compression. Vences and Rudomin use it to compress sequences of images [23], Wu et al. [24] improve upon Vences and Rudomin's approach, and Boucetta and Melkemi describe how to transform the RGB planes of a color image into more suitable spaces using a genetic algorithm [2].

Fang et al. investigate how to compress database information using GPUs to overcome the transfer overhead [7]. They employ a compression planner along with a cost model of the GPU to identify an optimal combination among nine different compression schemes and use a rule-based method to automatically prune the search space. They utilize fewer components than we do and, as in Kattan and Poli's work, each component is an entire compression algorithm.

Chaining whole compression algorithms, as is proposed in many of the above related works, is fundamentally different from chaining algorithmic components to build a compression algorithm, which is what we do. After all, the goal of a compression algorithm is to maximally reduce the number of bytes, which generally means that there are few exploitable patterns left in the output. This makes it difficult for the next compression algorithm in the chain to be effective. Our approach does not suffer from this problem. In fact, most of the algorithmic components we use do not reduce the number of bytes at all but transform the data to better expose patterns.

V. Evaluation Methodology

We evaluated all tested compressors on a compute node of the Maverick supercomputer at the Texas Advanced Computing Center. The node contains two 10-core Intel Xeon E5-2680 v2 Ivy Bridge processors running at 2.8 GHz with a 20 MB L3 cache and 128 GB of main memory. The operating system is CentOS 6.4. We used the icc compiler version 14.0.1 with the "-O3 -xhost" flags.

A. *Compression Tools*

We compare FPcrush in terms of compression ratio, compression throughput, and decompression throughput to three compressors from the literature: 1) pigz [16], a parallel version of gzip, 2) pbzip2 [13], a parallel version of bzip2, and 3) pFPC [15], a parallel version of FPC. The first two are widely used general-purpose compressors, i.e., they are not specifically designed for floating-point data. pFPC is a special-purpose compressor designed for double-precision floating-point data. It does not support single-precision data. Since our primary objective is to obtain a high compression ratio and the secondary objective is fast decompression, we use pigz with the "-c9 -p20" flags, pbzip2 with the "-9 -p20" flags, and pFPC with one million table entries, 20 threads, and 4096-element chunks, which are the recommended parameters.

B. *FPcrush Parameter Space and Randomization*

FPcrush is parameterizable along multiple dimensions. We studied all combinations of 2, 4, 8, 16, 32, and 64 generations in the GA, 1, 2, 3, 4, 5, 6, and 7-stage algorithms, and 100% (i.e., the entire input), 10%, 1%, and 0.1% segments. We use a fixed population size of 20 to match the number of cores in our system (hyper-threading is turned off on Maverick). Based on the results, we empirically selected sixteen generations, five-stage algorithms, and one percent segments as the baseline configuration for our performance evaluation, which yields good compression ratios and throughputs.

The genetic operations rely on a random-number generator. As a consequence, using different seeds can result in different synthesized algorithms even when otherwise using the same configuration and the same input. To lower the impact of the random seed, we repeated every experiment three times with three different seeds and present the results from the run that produced the median compression ratio. This should make the results more representative of what can be expected on average from our approach.

C. *Throughput Measurements*

For the special-purpose floating-point compressors pFPC and FPcrush, the timing measurements are performed by adding code to read a timer before and after the compression and decompression code sections. For the general-purpose compressors pbzip2 and pigz, we measure the runtime of compression and decompression when reading the input file from a disk cache in main memory and writing the output to /dev/null. In case of FPcrush, the compression time includes the time to select a representative data segment, running the genetic algorithm, and using the resulting best algorithm to compress the entire file. In all cases, the decompressed data are compared to the original to ensure that every bit is identical. This validation is not included in the timings.

D. *Input Files*

We use eight FPC data sets for our evaluation [14]. Each file consists of a binary sequence of IEEE 754 double-precision floating-point values. They encompass numeric results (num) and observational data (obs). For the single-precision experiments, we simply converted the double-precision files.

The following 4 data sets stem from 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

The following 4 data sets stem 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

Table I provides pertinent information about the double-precision inputs. The first two data columns list the size in megabytes and in millions of double-precision values. The middle column shows the percentage of values that are unique. The fourth column displays the first-order entropy of the values in bits. The last column expresses the randomness of each input in percent, i.e., it reflects how close the first-order entropy is to that of a truly random data set with the same number of unique values. We chose these files because they contain real-world data and are large enough to demonstrate the utility of our approach while making parameter-space evaluations tractable. Note that FPcrush tends to be more efficient on larger files, especially the segmentation, which is important as many real-world scientific applications produce files that are much larger than our test files.

TABLE I. INFORMATION ABOUT THE DOUBLE-PRECISION INPUTS

	size (megabytes)	doubles (millions)	unique values (percent)	1st order entropy (bits)	randomness (percent)
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

VI. Experimental Results

A. *Compression Ratios*

Table II shows the compression ratios on the 16 single- and double-precision files as well as the geometric mean for each compressor. FPcrush is run with the baseline configuration of five stages, 1% segments, and 16 generations. pFPC does not support single-precision data.

Except on obs_spitzer, FPcrush yields the highest compression ratio on all tested inputs, in particular also on num_plasma, the most compressible of the studied files. The benefits of compression range from 11% to over a factor of 10. pbzip2 compresses the obs_spitzer file 26% and 41% better, presumably due to its use of a block-sorting algorithm, which is relatively slow and not synthesizable from the components included in FPcrush.

The results in Table II demonstrate that FPcrush is able to synthesize effective floating-point compression algorithms. In fact, it often yields never-before-described algorithms that compress better than some of the best available compressors.

B. *Decompression Speed*

Table III lists the decompression throughput in megabytes per second on the 16 inputs as well as the geometric mean. Again, FPcrush refers to the baseline configuration with five stages, 1% segments, and 16 generations.

The automatically synthesized decompression algorithms deliver the highest throughput on each tested file. The geometric mean is over one gigabyte per second, which is 5.4 and 5.7

TABLE II. COMPRESSION RATIOS (THE HIGHEST SINGLE- AND DOUBLE-PRECISION COMPRESSION RATIOS ARE SHADED)

		geomean	num_brain	num_comet	num_control	num_plasma	obs_error	obs_info	obs_spitzer	obs_temp
double	pigz	1.206	1.064	1.160	1.057	1.608	1.447	1.157	1.228	1.035
	pbzip2	1.460	1.043	1.173	1.029	5.670	1.331	1.218	1.746	1.023
	pFPC	1.440	1.148	1.151	1.038	7.042	1.542	1.215	1.022	0.997
	FPcrush	1.665	1.194	1.285	1.127	10.547	1.672	1.403	1.237	1.114
single	pigz	1.524	1.113	1.117	1.043	8.652	1.338	1.327	1.391	1.049
	pbzip2	1.510	1.113	1.117	1.043	8.781	1.337	1.219	1.389	1.048
	FPcrush	1.667	1.302	1.203	1.157	10.189	1.613	1.606	1.106	1.128

TABLE III. DECOMPRESSION THROUGHPUT [MB/S] (THE HIGHEST SINGLE- AND DOUBLE-PRECISION THROUGHPUTS ARE SHADED)

		geomean	num_brain	num_comet	num_control	num_plasma	obs_error	obs_info	obs_spitzer	obs_temp
double	pigz	217.0	200.4	212.4	214.6	269.2	236.4	206.5	196.0	208.7
	pbzip2	122.9	67.3	121.8	66.1	330.0	200.1	168.0	45.4	190.8
	pFPC	786.3	883.2	808.0	791.2	1112.4	781.3	541.8	889.1	618.0
	FPcrush	1247.5	1494.6	1464.5	1425.4	1231.2	1195.4	852.7	1335.5	1121.7
single	pigz	206.3	199.6	224.6	204.1	214.0	218.4	198.6	191.4	201.7
	pbzip2	170.6	210.7	207.0	141.1	249.5	195.9	142.0	102.1	164.6
	FPcrush	1121.7	1231.3	1214.3	1387.0	964.0	849.2	1333.1	1148.2	964.6

TABLE IV. COMPRESSION THROUGHPUT [MB/S] (THE HIGHEST SINGLE- AND DOUBLE-PRECISION THROUGHPUTS ARE SHADED)

		geomean	num_brain	num_comet	num_control	num_plasma	obs_error	obs_info	obs_spitzer	obs_temp
double	pigz	384.3	423.8	362.3	451.0	537.1	264.9	381.2	300.6	421.8
	pbzip2	75.6	87.2	101.1	87.3	20.8	99.1	73.8	113.4	80.5
	pFPC	1080.4	1370.5	1262.4	1149.9	1310.4	1073.5	596.7	1395.4	796.7
	FPcrush	122.0	132.0	130.8	149.8	111.7	130.3	74.9	144.1	120.9
single	pigz	304.5	358.8	410.8	419.7	401.8	143.4	289.8	212.5	336.5
	pbzip2	70.3	85.8	87.6	74.6	22.7	89.2	73.0	96.2	74.7
	FPcrush	134.2	133.3	142.5	166.1	127.1	115.2	89.1	178.7	142.8

times faster than pigz and 6.6 and 10.1 times faster than pbzip2. pFPC is outperformed by a factor of 1.6. Note that each of these compressors utilizes all 20 CPU cores.

For comparison, we also measured the memory copy throughput of the built-in memcpy function on the same data. It ranges from 3358 MB/s to 8707 MB/s. In other words, simply copying the data from one memory location to another is 2.4 to 6.5 times faster than the decompression speed of FPcrush's algorithms on our machine.

Clearly, the synthesized algorithms decompress quickly and compress well. In fact, the benefit in decompression throughput is higher than the benefit in compression ratio.

C. Compression Speed

Table IV shows the compression throughput in megabytes per second on the 16 input files and the geometric mean. As before, FPcrush refers to the baseline configuration with five stages, 1% segments, and 16 generations. Note that the measured runtime of FPcrush includes the time to identify the best segment, the time to run the genetic algorithm on this segment, and the time to compress the entire file using the best algorithm the GA found.

pFPC compresses the double-precision files the fastest. Since it does not support single-precision data, pigz performs best on those files. Surprisingly, FPcrush delivers well over 100 MB/s compression throughput even though it first has to determine a suitable algorithm. This throughput is sufficient for real-time compression on a gigabit-per-second channel.

On average, FPcrush is nine times slower than pFPC and two to three times slower than pigz, but it is 1.6 and 1.9 times faster than pbzip2. In addition to being faster than pbzip2 on each tested input, it also compresses better and decompresses more quickly, which underscores the benefit of our approach.

Figure 2 illustrates the relative breakdown of FPcrush's compression time on the double-precision files when using five stages, 1% segments, and 16 generations. The bottom part of each bar shows the time to find the best segment, the middle part shows the time to run the genetic algorithm on this segment, and the top part shows the time to compress the entire file using the best found algorithm.

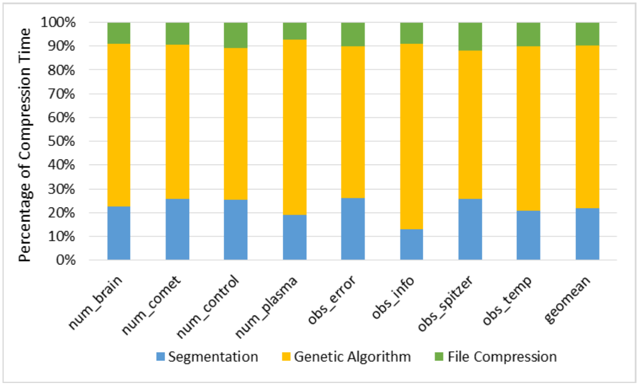

Fig. 2. Breakdown of the double-precision compression time

Running the genetic algorithm dominates and takes on average 68% of the overall compression time. Determining a good segment takes 22% of the runtime. It is relatively slow because of calculating the entropy, which requires slow transcendental floating-point operations whereas the compression is performed exclusively with fast integer operations. The actual compression takes about 10% of the overall runtime. Combined with the throughput results from Table IV, we find the true compression to be about as fast as the decompression (cf. Table III). The synthesis (the genetic algorithm) plus the segmentation slow down compression by a factor of ten.

TABLE V. SYNTHESIZED ALGORITHMS

	customized double-precision algorithm	customized single-precision algorithm
num_brain	| ROT1 ROT7 ROT1 DIM8 LZ5	| DIM4 ROT1 LZ6 DIM12 LZ7
num_comet	BIT ZE RLE | ROT2 LZ6	| DIM12 SMS DIM7 SMS LZ4
num_control	ROT2 LVx BIT DIM64 | LZ4	LVs BIT RLE | ROT7 LZ5
num_plasma	ROT1 SMS LZ2 LZ6 | LZ3	LVs LZ7 | DIM8 LVx LZ4
obs_error	LZ3 | DIM64 ROT1 DIM3 LZ6	DIM2 | NUL LZ7 DIM12 LZ5
obs_info	LVs | DIM8 LZ3 ROT2 LZ7	| NUL DIM4 DIM2 ROT5 LZ3
obs_spitzer	LZ1 LZ2 ZE BIT | LZ5	DIM32 BIT ROT5 DIM32 | LZ4
obs_temp	BIT ROT4 DIM64 | LVx LZ4	DIM8 LVs BIT ROT2 | LZ4

D. Synthesized Algorithms

Table V illustrates the most effective algorithm FPcrush synthesized for each tested input when using five stages, 1% segments, and 16 generations. The component names follow the description in Section 2.

While not easy to understand, it is obvious that the algorithms differ substantially from one input to another and, perhaps more surprisingly, even between the single- and double-precision versions of the same input. However, this is at least in part due to performing an imperfect search using an imperfect segment. These imperfections are reflected, for example, in the double-precision algorithm for num_brain, which contains three ROT components in a row that should be replaced by an equivalent single ROT component. Adding a post-processing step to FPcrush could identify and eliminate such artifacts, which might improve the synthesis results, the convergence of the GA, and the throughput of the algorithms.

Other observations of note include, for example, that two of the single-precision algorithms contain a NUL component, meaning that they really only have four stages. SMS, the other mutator, also occurs twice. All shufflers (BIT, ROT, and DIM) are very frequent. As mentioned, ROT is needed to adjust the hash function of the LZ components. Both the subtraction- and the XOR-based LV predictor appear in several algorithms, through subtraction dominates. Finally, all reducers are employed. RLE and ZE occur twice. At least one version of LZ is included in every algorithm with up to three in a single algorithm. Whereas all parameters of LZ are used, only parameters 3 through 7 are frequent with 4 being the most frequent. For DIM, the parameter 5 is not used. For ROT, parameters 3 and 6 do not occur.

Clearly, not all components are equally important. Eliminating some components/parameters would speed up the genetic algorithm and the synthesis, but it is unknown whether these components are useful on other files. A more extensive study is needed to determine if they are truly unnecessary.

The Cut never appears at the end, indicating that it is useful to eventually process the data at byte granularity. Note that most of the employed components are not reducers, i.e., do not compress the data, demonstrating that transforming the values to make them more amenable for the reducers is paramount in an efficient compression algorithm.

E. Segment Size

The key novelty of FPcrush is its ability to synthesize customized compression algorithms in real-time. This subsection studies the use of segments, i.e., one of the main techniques that make this possible. Since the single-precision results exhibit the same trends, we only present double-precision results. We use exhaustive search as the baseline, which is very slow. Hence, we can only show results for three-stage algorithms, the largest chain length for which the exhaustive search completes within the 12-hour job limit on Maverick.

Figure 3 shows the throughput results in megabytes per second on a log-log plot. We ran the exhaustive search on the entire input (100%) as well as with segments that are 10%, 1%, and 0.1% as long as the complete inputs.

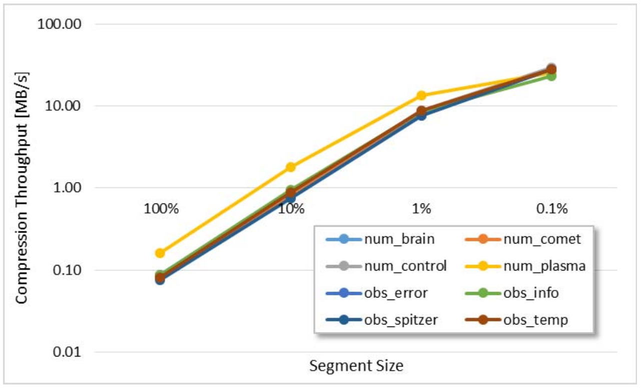

Fig. 3. Double-precision compression throughput of exhaustive search with three stages as a function of the segment size

The results show roughly linear scaling with the segment size down to one percent. In fact, all eight inputs exhibit some superlinear scaling in this range because the entire working set does not fit in the processor's L3 cache, but when using sufficiently small segments it does. The scaling drops off when going to 0.1% segments because the search becomes so fast that other work starts to dominate, most notably the ultimate compression of the entire input with the best algorithm. For exhaustive search, the workload is exponential in the length of the chains. Hence, even smaller segments are likely to be beneficial with four or more stages. Having said that, care should be taken not to make the segments too short. For example, the

0.1% segment of obs_info is only 18 kilobytes long. We empirically determined that at least a few kilobytes are necessary to obtain compression algorithms that work well on the full inputs. This means very small segment percentages are only prudent for large files to speed up the synthesis.

Using segments is clearly an effective approach to reduce the algorithm synthesis time and therefore to increase the compression throughput. However, using segments lowers the compression ratio in cases where no representative segment can be found. To investigate the magnitude of this potential problem, Table VI shows the achieved compression ratio relative to that of exhaustive search on the whole inputs.

TABLE VI. HIGHEST COMPRESSION RATIO OF EXHAUSTIVE SEARCH WHEN USING SEGMENTS RELATIVE TO USING THE ENTIRE FILE

	100%	10%	1%	0.1%
num_brain	1.00	1.00	1.00	1.00
num_comet	1.00	1.00	1.00	1.00
num_control	1.00	0.96	0.99	1.00
num_plasma	1.00	1.00	1.00	0.53
obs_error	1.00	1.00	0.99	0.99
obs_info	1.00	1.00	1.00	0.96
obs_spitzer	1.00	1.00	1.00	0.97
obs_temp	1.00	1.00	1.00	1.00

10% and 1% segments yield essentially the same compression ratio on our eight double-precision files as using the whole files does. 0.1% segments are also quite good except on num_plasma, where no representative segment was found. Nevertheless, with 0.1% segments, FPcrush is still able to determine the best algorithm in its search space for half of the studied inputs. Interestingly, on num_control, smaller segments yield better results than larger segments. This is possible because longer segments may be less representative of the entire input than the finer-grained shorter segments.

F. Number of Generations

This subsection studies the number of generations in the genetic algorithm, another key parameter that greatly affects the synthesis speed. We again only present double-precision results and use exhaustive search as the baseline.

Figure 4 shows the throughput on a log-log plot for the genetic algorithm with 64, 32, 16, 8, 4, and 2 generations. In all cases, the entire files were used rather than segments.

The throughput scales well with decreasing numbers of generations. num_plasma is again an outlier and yields the highest throughput because it is more compressible than the other files. Clearly, reducing the number of generations is another effective way to speed up the algorithm synthesis. However, doing so can hurt the quality of the best found algorithm, as the results in Table VII illustrate.

As expected, Table VII shows that the algorithm quality tends to decrease with fewer generations. While not severe for three-stage chains, where the genetic algorithm quickly finds the best compression algorithm in the search space, the quality degrades more for longer chains of components. Since the search space is exponential in the length of the chain, longer chains require more generations for the genetic algorithm to find high-quality solutions. Since exhaustive search is intractable for long chains, using a genetic algorithm (or any other fast search method) not only drastically speeds up the search for effective multi-stage compression algorithms but is essential in that it makes such searches possible in the first place.

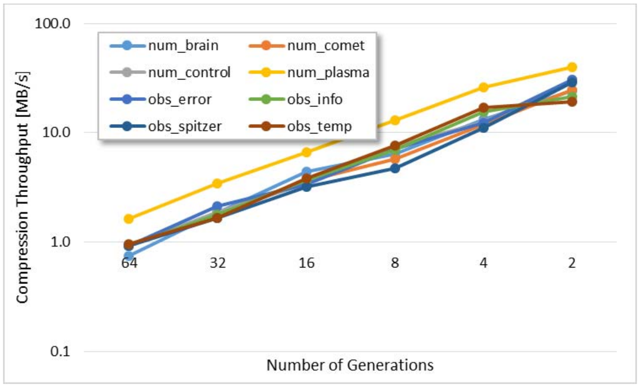

Fig. 4. Compression throughput of the genetic algorithm with three stages and no segments as a function of the number of generations

TABLE VII. HIGHEST COMPRESSION RATIO WITH THREE STAGES FOR VARIOUS NUMBERS OF GENERATIONS RELATIVE TO EXHAUSTIVE SEARCH

	64	32	16	8	4	2
num_brain	0.99	0.99	1.00	0.99	0.99	0.99
num_comet	1.00	0.97	0.97	0.91	0.96	0.93
num_control	0.99	0.98	0.98	0.98	0.98	0.91
num_plasma	1.00	1.00	1.00	0.97	0.92	0.89
obs_error	0.99	1.00	0.99	0.93	0.98	0.92
obs_info	1.00	1.00	0.99	0.95	0.96	0.94
obs_spitzer	0.97	0.96	0.96	0.94	0.94	0.92
obs_temp	0.99	0.95	0.98	0.99	0.98	0.92

G. Number of Stages

Table VIII shows the compression ratios on the eight double-precision files when running the genetic algorithm for 16 generations on 1% segments with various numbers of stages.

Expectedly, longer chains tend to perform better as they can express supersets of the algorithms with fewer stages. However, because the GA-based search is imperfect, this is not always the case. For example, on obs_info, the best found four- and six-stage algorithms are worse than the best three-stage algorithm. The geometric mean compression ratio climbs steadily up to five stages, beyond which the increase is lower and even drops at six stages. Note that longer chains result in lower compression and decompression throughput. Five-stage algorithms seem to yield good compression ratios without overly burdening the throughput. Nevertheless, if a higher throughput is desired, using fewer stages is an obvious approach to achieve that. Since more stages do help on some inputs and the user does not know a priori how many stages will suffice, we evaluated and recommend using FPcrush with five stages or thereabouts.

TABLE VIII. COMPRESSION RATIO USING 16 GENERATIONS WITH 1% SEGMENTS AS A FUNCTION OF THE NUMBER OF ALGORITHM STAGES

	2	3	4	5	6	7
num_brain	1.19	1.19	1.17	1.19	1.20	1.20
num_comet	1.27	1.30	1.31	1.29	1.27	1.33
num_control	1.10	1.11	1.13	1.13	1.12	1.13
num_plasma	9.37	8.51	10.87	10.55	10.56	11.27
obs_error	1.36	1.53	1.49	1.67	1.69	1.68
obs_info	1.38	1.40	1.24	1.40	1.24	1.41
obs_spitzer	1.21	1.23	1.27	1.24	1.23	1.24
obs_temp	1.09	1.10	1.11	1.11	1.12	1.11
geomean	1.58	1.60	1.63	1.66	1.64	1.69

H. Other Parameters

We did not present an evaluation of the population size used by the genetic algorithm, the size of the hash tables in the LZ components, and the chunk size, in favor of more detailed measurements and analysis of more important parameters. We used a fixed population size of 20 to match the number of cores in our system. This parameter is somewhat uninteresting as larger populations hardly improve the final algorithms but make the synthesis slower. We selected a constant hash-table size of 65,536 words to maximize the compression ratio while mostly hitting in the L2 cache. Larger tables increase the average compression ratio only a little but substantially decrease the compression and decompression throughput while smaller tables result in significantly lower compression ratios. We chose a fixed chunk size of 131,072 words for similar reasons, i.e., to fully exploit the L3 cache. Larger chunks only improve the compression ratio a little while significantly lowering the compression and decompression throughput. Smaller chunks hurt the compression ratio considerably.

VII. SUMMARY AND CONCLUSIONS

This paper describes a high-speed approach to automatically synthesize data compression and matching decompression algorithms. The key novelty of our work is to demonstrate that superior compression algorithms can, in fact, be synthesized in real-time. To the best of our knowledge, we are the first to show that this is doable at all and to present a combination of techniques that make it possible. This combination is essential as the parallel genetic algorithm, the segmentation approach, and the linear-time components are only fast enough when used together to achieve real-time synthesis.

We implemented our approach in the FPcrush tool, which is based on a set of algorithmic components that can be chained to construct sophisticated compression algorithms. FPcrush employs a genetic algorithm to quickly search for the most effective chains of components, i.e., algorithms, and uses small representative segments of the input data to further accelerate the search. Together, these techniques make our compression-algorithm-synthesis tool faster than the parallel bzip2 compressor while compressing better in most cases. In spite of FPcrush's synthesis overhead, even compression operates at over 100 megabytes per second.

There are several avenues for future work. For instance, other fast search algorithms could be tried, as could other mechanisms for identifying representative segments such as sampling. The presented study could also be expanded to include more inputs, larger inputs, and non-floating-point inputs. Moreover, one could make it possible to trade off compression ratio and throughput by allowing the end-user to choose which components to include, what table and population size to use, etc. Another interesting idea is to seed the GA with promising algorithms rather than starting with a random population. To support streaming data in FPcrush, the approach for determining a good segment would have to be changed to not require all the data to be present or available.

We believe the FPcrush approach to be applicable to other domains. All that is needed is for an expert to develop transformations and inverses thereof for the new domain so that corresponding components can be added to the database. FPcrush will then automatically incorporate the new components if they turn out to be useful. Aside from its direct application to data compression, we hope that our work will inspire others to build similar systems for other environments.

ACKNOWLEDGMENT

This work was supported by the U.S. National Science Foundation under grants 1141022, 1217231, 1406304, and 1438963 as well as a REP grant from Texas State University. The authors acknowledge the Texas Advanced Computing Center for providing the HPC resources used in this study.

REFERENCES

[1] T. Bicer, J. Yiny, D. Chiuz, G. Agrawal, and K. Schuchardt. "Integrating Online Compression to Accelerate Large-Scale Data Analytics Applications." *International Parallel and Distributed Processing Symposium*. 2013.

[2] A. Boucetta and K.E. Melkemi. "DWT Based-Approach for Color Image Compression Using Genetic Algorithm." *5th International Conference on Image and Signal Processing*, pp. 476-484, June 2012.

[3] M. Burtscher and P. Ratanaworabhan. "FPC: A High-Speed Compressor for Double-Precision Floating-Point Data." *IEEE Transactions on Computers*, 58(1):18-31. 2009.

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

[5] M. Burtscher and N.B. Sam. "Automatic Generation of High-Performance Trace Compressors." *International Symposium on Code Generation and Optimization*, pp. 229-240. 2005.

[6] D. Chen, Y.-J. Chiang, N. Memon, and X. Wu. "Lossless Geometry Compression for Steady-state and Time-varying Irregular Grids." *IEEE Symposium on Visualization*, pp. 275-282. 2006.

[7] W. Fang, B. He, and Q. Luo. "Database Compression on Graphic Processors." *Proceedings of the VLDB Endowment*, 3(1-2):670-680. 2010.

[8] R. Filgueira, D.E. Singh, A. Calderón, and J. Carretero. "CoMPI: Enhancing MPI-based Applications Performance and Scalability Using Run-Time Compression." *EUROPVM/MPI*. 2009.

[9] R. Filgueira, D.E. Singh, J. Carretero, A. Calderón, and F. Garcia. "Adaptive-CoMPI: Enhancing MPI-based Applications - Performance

and Scalability by using Adaptive Compression." *International Journal of High Performance Computing Applications*, 25(1):93-114. 2011.

[10] D.E. Goldberg. "Genetic algorithms in search, optimization, and machine learning." *Addison Wesley*. 1989.

[11] J.H. Holland. "Adaptation in natural and artificial systems." *University of Michigan press*, 1:97. 1975.

[12] W.H. Hsu and A.E. Zwarico. "Automatic synthesis of compression techniques for heterogeneous files." *Software: Practice and Experience*, 25(10):1097-1116. 1995.

[13] http://compression.ca/pbzip2/

[14] http://cs.txstate.edu/~burtscher/research/datasets/FPdouble/

[15] http://users.ices.utexas.edu/~burtscher/research/pFPC/

[16] http://zlib.net/pigz/

[17] J. Jenkins, I. Arkatkar, S. Lakshminarasimhan, D.A. Boyuka II, E.R. Schendel, N. Shah, S. Ethier, C.-S. Chang, J. Chen, H. Kolla, S. Klasky, R. Ross, N.F. Samatova. "ALACRITY: Analytics-Driven Lossless Data Compression for Rapid In-Situ Indexing, Storing, and Querying." *Transactions on Large-Scale Data- and Knowledge-Centered Systems X, Lecture Notes in Computer Science*, vol. 8220, pp. 95-114. 2013.

[18] A. Kattan and R. Poli. "Evolutionary synthesis of lossless compression algorithms with GP-zip3." *IEEE Congress on Evolutionary Computation*, 1(8):18-23. 2010.

[19] P. Lindstrom. "Fixed-Rate Compressed Floating-Point Arrays." *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674-2683. December 2014.

[20] P. Lindstrom and M. Isenburg. "Fast and Efficient Compression of Floating-Point Data." *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245-1250. 2006.

[21] S.K. Mitra, C. A. Murthy, and K. Malay. "Technique for Fractal Image Compression using Genetic Algorithm." *IEEE Transactions on Image Processing*, pp. 586-593. 1998.

[22] E. R. Schendel, Y. Jin, N. Shah, J. Chen, C. S. Chang, S-H. Ku, S. Ethier, S. Klasky, R. Latham, R. B. Ross, and N. F. Samatova. "ISOBAR preconditioner for effective and high-throughput lossless data compression." *28th Annual IEEE International Conference on Data Engineering,* pp. 138-149. 2012.

[23] L. Vences and I. Rudomin. "Genetic Algorithms for Fractal Image and Image Sequence Compression." *Comptacion Visual*. 1997.

[24] M.S. Wu, J.H. Jeng, and J.G. Hsieh. "Schema genetic algorithm for fractal image compression." *Engineering Applications of Artificial Intelligence*, 20(4):531-538. June 2007.

[25] M.S. Wu and Y.L. Lin. "Genetic algorithm with a hybrid select mechanism for fractal image compression." *Digital Signal Processing*, 20(4):1150-1161. July 2010.

[26] A. Yang, H. Mukka, F. Hesaaraki, and M. Burtscher. "MPC: A Massively Parallel Compression Algorithm for Scientific Data." *IEEE Cluster Conference*, pp. 381-389. September 2015.

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