SPDP: An Automatically Synthesized Lossless Compression Algorithm for Floating-Point Data

Steven Claggett, Sahar Azimi, and Martin Burtscher

Department of Computer Science
Texas State University
San Marcos, TX 78666, USA

***Abstract**:* Scientific computing produces, transfers, and stores massive amounts of single- and double-precision floating-point data, making this a domain that can greatly benefit from data compression. To gain insight into what makes an effective lossless compression algorithm for such data, we generated over nine million algorithms and selected the one that yields the highest compression ratio on 26 datasets. The resulting algorithm, called SPDP, comprises four data transformations that operate exclusively at word or byte granularity. Nevertheless, SPDP delivers the highest compression ratio on eleven datasets and, on average, outperforms all but one of the seven compared compressors. An analysis of SPDP's internals reveals how to build effective compression algorithms for scientific data.

1. Introduction

Scientific computing applications often produce and transfer large amounts of floating-point data. For example, many simulations exchange data between processing nodes and with mass storage devices after every time step. Some large datasets have to be sent to other locations for additional processing, analysis, or visualization. Moreover, long-running programs regularly save checkpoints to disk so that they can resume execution from the most recent checkpoint after a crash. It is well known that compression can reduce the amount of data that needs to be transferred and stored in these and other situations.

Many users of scientific computing take advantage of frameworks like HDF5 for managing their data [7]. Some of these frameworks support compression "filters". However, such filters are only likely to be employed if they are effective. Moreover, they need to be lossless so that they can safely be used in any setting. Depending on the application, scientific data are typically stored in either single- or double-precision IEEE 754 floating-point format. Users do not want to have to select different filters depending on the data's precision. Rather, they would like a single algorithm that compresses both types well.

To identify an effective lossless compression algorithm that meets this criterion, we used our CRUSHER framework to systematically generate over nine million algorithm candidates from a set of 48 algorithmic components. Each component implements a data transformation and can operate at word and byte granularity. CRUSHER then performed an exhaustive search to determine the best four-component algorithm in this search space for a suite of 13 single- and 13 double-precision datasets. The goal of the work presented in this paper is to answer the following two research questions. 1) *Can we gain insight and learn from the resulting algorithm, for example, how to best handle mixed single/double datasets?* 2) *Can a competitive compression algorithm be created from data transformations that only process data at word and byte granularity but not at bit granularity?*

We named the resulting algorithm SPDP, which is an abbreviation for "Single Precision

Double Precision". It is brand new and not similar to prior compression algorithms. SPDP delivers the highest compression ratio on eleven of the 26 tested datasets. Only Zstd performs better. On average, SPDP outperforms Blosc, bzip2, FastLZ, LZ4, LZO, and Snappy by at least 30% in terms of compression ratio. However, it tends to be slower.

This paper makes the following contributions. 1) We systematically search 9,400,320 combinations of four algorithmic components to determine the best compression algorithm within this space. 2) We analyze the resulting sequence of components to gain insight into why it works well and how to handle mixed types of inputs. 3) We present a previously unknown algorithm (SPDP) that compresses floating-point data well even without any bit-granularity coders. 4) We compare many compression algorithms that are available as HDF5 filters in terms of compression ratio, compression throughput, and decompression throughput. SPDP is freely available as a standalone compressor and as an HDF5 filter [6].

The rest of this paper is organized as follows. Section 2 describes our algorithm synthesis approach. Section 3 summarizes related work. Section 4 provides an overview of the system, compressors, and datasets we use. Section 5 presents and discusses the experimental results. Section 6 concludes with a summary and future work.

2. Algorithm Synthesis

To be able to systematically search for effective compression algorithms, we built a framework called CRUSHER for automatically synthesizing compressors and the corresponding decompressors. It is based on a library of interoperable *algorithmic components*. These components are the result of a thorough analysis of preexisting compression algorithms. In particular, we broke many prior algorithms down into their constituent parts and generalized them. This yielded a number of algorithmic components for building data models and coders. We then implemented these components using a common interface such that 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 combine (chain) the components in any way, allowing for the generation of a large number of compression-algorithm candidates from a small set of components. CRUSHER uses exhaustive search to automatically determine the best chain of components (aka compression algorithm) in its search space for a given set of input data. Importantly, for each algorithmic component, the framework includes an inverse that performs the opposite transformation. Thus, for any chain of components, it is straightforward to synthesize the matching decompressor.

2.1. Algorithmic Components

This section describes the 48 algorithmic components available to CRUSHER for synthesizing compression algorithms. Many of them are generalizations or approximations of data transformations extracted from prior algorithms. Each component takes a sequence of values as input (i.e., an array), transforms it, and outputs the transformed sequence.

The **NUL** component simply outputs the input sequence. It ensures that chains with *n* components can also represent all possible algorithms with fewer than *n* components.

The **BIT** component groups the values into chunks of as many values as there are bits per value. It then transforms each chunk by 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 **DIM*n*** component takes a parameter n that specifies the dimensionality 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. We use n = 2, 4, 8, and 12.

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 that rotate the bits by one eighth to seven eighths of the word size in bits.

The **LNV*kn*** component takes two parameters. It subtracts the last n^{th} value from the current value and emits the residual. If k = 's', arithmetic subtraction is used. If k = 'x', bitwise subtraction (xor) is used. In both cases, we tested n = 1, 2, 3, 4, 8, 12, 16, 32, and 64.

None of the above components change the size of the data blocks. The next three components are the only ones that can reduce the length of a data block, i.e., compress it.

The **ZE** component operates on chunks of eight values and emits a bitmap that contains a bit for every value in the chunk. Each bit indicates whether the corresponding value is zero or not. Following the bitmap, ZE emits the non-zero values from the chunk.

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 then the non-repeating values.

The **LZ*ln*** component implements a variant of the LZ77 algorithm [13]. It incorporates tradeoffs that make it more efficient than other LZ77 versions on hard-to-compress data and operates as follows. It uses a 32768-entry hash table to identify the l most recent prior occurrences of the current value. Then it checks whether the n values immediately preceding those locations match the n values just before the current location. If they do not, only 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 after that location. The length of the matching substring is emitted and the component advances by that many values. We consider n = 3, 4, 5, 6, and 7 combined with l = 'a', 'b', and 'c', where 'a' = 1, 'b' = 2, and 'c' = 4, which yields fifteen LZ*ln* components.

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. Every algorithm produced by CRUSHER contains a Cut, which is included because it may be more effective to perform none, some, or all of the compression at byte rather than word granularity.

Each component includes a corresponding inverse component that performs the reverse data transformation. By chaining the inverse components in the opposite order, CRUSHER can automatically synthesize the matching decompression algorithm for any chain of components, i.e., for any compression algorithm it can generate.

Due to the Cut, we need two versions of each component and its inverse, one for words (4-byte values) and one for bytes. We implemented all components using C++ templates to facilitate the generation of these versions. There are no components that emit data at bit granularity such as Huffman or arithmetic coders. Each component exclusively uses integer operations and works on an integer representation, i.e., each floating-point value is copied bit by bit into an integer. All included components run in linear time. We excluded more complex components such as move-to-front and block-sorting components to make the synthesis faster, which has to evaluate millions of algorithms.

Not counting the Cut, CRUSHER has 48 components at its disposal, 17 of which are able to reduce the length of the data. The purpose of the remaining 31 components is to transform the values in such a way that the reducers become maximally efficient. Chains with more than four components have the potential for better compression but would make the search for the best algorithm nearly intractable. For an algorithm with k stages, i.e., a chain with k components, the search space encompasses $(k+1) \cdot 48^{k-1} \cdot 17$ possibilities because there are $k+1$ locations for the Cut, $k-1$ stages that can each hold any one of the 48 components (duplicates are allowed), and a final stage that can hold any one of the 17 reducers. In our case, this amounts to 9,400,320 possible four-stage algorithms.

3. Related Work

This section describes prior work on synthesizing compression algorithms. We first used CRUSHER to create a massively-parallel floating-point compression algorithm for GPUs [12]. That work employs some of the same algorithmic components. However, it only utilizes components that can easily be parallelized. SPDP does not have this limitation, which is why it almost always compresses better, in some cases by a large margin.

Then, we modified CRUSHER itself by adding a fast heuristic- and sampling-based parallel search algorithm to accelerate the synthesis [2]. This makes it possible to generate compression algorithms in real-time that are tailored to each dataset, i.e., to compress each dataset with a potentially unique algorithm. Whereas this customization yields higher compression ratios on some inputs, the fast search tends to miss some good algorithms in the search space. On average, the compression ratio is 12% lower on the double- and 20% lower on the single-precision datasets compared to SPDP, whose synthesis took days.

None of the remaining related works described below are designed for floating-point data. Instead, they target integers, program execution traces, heterogeneous files, images, and databases. Hence, we do not compare SPDP to these approaches.

We utilized CRUSHER to generate integer compression algorithms that are space-probe friendly [3]. That work also partially uses the same algorithmic components. However, it only employs components that require very little state (small or no tables/dictionaries) as most space probes only contain a small amount of (radiation-hardened) main memory.

In much earlier work, we presented TCgen, a tool to generate customized trace compressors from a user-provided configuration of one or more predictors [1]. TCgen then translates this description into C source code that is optimized for the specified trace format and predictors (components). CRUSHER supports many more 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 [9]. They group similar data chunks together and label each group with the best compression algorithm for its chunks. Similarly, Hsu and Zwarico present an automatic synthesis technique for compressing heterogeneous files [8]. Each chunk of data is compressed using a different algorithm, which is determined using a statistical method. Note that they combine chunks of the input data that were potentially compressed with different algorithms whereas we combine components to form a single compression algorithm that is used throughout.

The same distinction applies to Mitra et al., who propose a methodology for compressing fractal images [10]. 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. The purpose of this classification is to obtain higher compression ratios and to reduce the encoding time. Wu and Lin use a similar approach with three classes [11].

Fang et al. investigate how to compress database information to minimize the CPU/GPU transfer overhead [4]. They use a compression planner and a cost model to identify an optimal combination among nine different compression schemes and employ 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 some of the above work, 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 a 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 but transform the data to better expose patterns (cf. Section 2.1).

4. Methodology

4.1. Compressors

For our evaluation, we selected the compressors from the list of HDF5 filters [6] that are lossless and serial and that we could get to compile and run. In addition to SPDP, this is Blosc, bzip2, FastLZ, LZ4, LZO, Snappy, and Zstd. We use Blosc without any pre-conditioner as we found them to lower the compression ratio on our datasets. Where possible, we run each compressor with its fastest as well as its best compressing configuration. In all cases, we evaluate the standalone compressor without employing HDF5.

4.2. Measurements

For each tested compressor and configuration, we report the compression ratio, the compression throughput, and the decompression throughput. The compression ratio is the number of bytes in the uncompressed dataset divided by the number of bytes in the compressed dataset. Hence, higher ratios are better. To measure the runtime, we timed the tools' execution on the command line. To exclude the disk speed from the timing measurements, the input datasets were cached in main memory and the outputs were written to /dev/null. To obtain the throughput, we divided the original dataset size by the measured runtime. We report throughputs rather than runtimes because throughputs are independent of the dataset size and more amenable to averaging. Moreover, throughput is also a higher-is-better metric. Each experiment was conducted five times and the median throughput is reported. For each tool, we verified that the decompressed output matches the original dataset exactly.

4.3. Datasets

We use the thirteen FPC datasets for our evaluation [5]. They include observational data (obs), numeric results (num), and MPI messages (msg). Table 1 provides information about each dataset. The first two numeric 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 dataset in percent, i.e., it reflects how close the first-order entropy is to that of a truly random dataset with the same number of unique values. For the single-precision versions, we simply converted the double-precision data.

Table 1: Information about the double-precision datasets

	size	doubles	unique values	1st order	randomness
	(megabytes)	(millions)	(percent)	entropy (bits)	(percent)
msg_bt	254.0	33.30	92.9	23.67	95.1
msg_lu	185.1	24.26	99.2	24.47	99.8
msg_sp	276.7	36.26	98.9	25.03	99.7
msg_sppm	266.1	34.87	10.2	11.24	51.6
msg_sweep3d	119.9	15.72	89.8	23.41	98.6
num_brain	135.3	17.73	94.9	23.97	99.9
num_comet	102.4	13.42	88.9	22.04	93.8
num_control	152.1	19.94	98.5	24.14	99.6
num_plasma	33.5	4.39	0.3	13.65	99.4
obs_error	59.3	7.77	18.0	17.80	87.2
obs_info	18.1	2.37	23.9	18.07	94.5
obs_spitzer	189.0	24.77	5.7	17.36	85.0
obs_temp	38.1	4.99	100.0	22.25	100.0

4.4. System and Compiler

We compiled the tested codes with gcc/g++ 5.3.1 using the "-O3 -march=native" flags. We measured the runtime of the compressors on a system with dual 10-core Xeon E5-2687W v3 CPUs running at 3.1 GHz. Each core has separate 32 kB L1 caches, a unified 256 kB L2 cache, and the cores on a socket share a 25 MB L3 cache. The host memory size is 128 GB and has a peak bandwidth of 68 GB/s. The operating system is Fedora 23.

5. Experimental Results

The following subsections present the results of our evaluation. The first subsection analyzes the structure of the synthesized SPDP algorithm. The remaining subsections compare it to the other compressors in terms of compression ratio, compression throughput, and decompression throughput. Whenever we mention an average, it is the geometric mean.

5.1. Synthesized Algorithm

SPDP, the best-compressing four-component algorithm for our datasets in CRUSHER's 9,400,320-entry search space is **LNVs2 | DIM8 LNVs1 LZa6**. Whereas there has to be a reducer component at the end, none appear in the first three positions, i.e., CRUSHER generated a three-stage data model followed by a one-stage coder. This result shows that chaining whole compression algorithms, each of which would include a reducer, is not beneficial. Also, the Cut appears after the first component, so it is important to first treat the data at word granularity and then at byte granularity to maximize the compression ratio.

The **LNVs2** component at the beginning that operates at 4-byte granularity is of particular interest. It subtracts the second-previous value from the current value in the sequence and emits the residual. This enables the algorithm to handle both single- and double-precision data well. In case of 8-byte doubles, it takes the upper half of the previous double and subtracts it from the upper half of the current double. Then it does the same for the lower halves. The result is, except for a suppressed carry, the same as computing the difference sequence on 8-byte values. In case of 4-byte single-precision data, this component also computes the difference sequence, albeit using the second-to-last rather than the last value. If the values are similar, which is where difference sequences help, then the second-previous value is also similar and should yield residuals that cluster around zero as well. This observation answers our first research question. *We are able to learn from the synthesized algorithm, in this case how to handle mixed single/double-precision datasets*.

The **DIM8** component after the **Cut** separates the bytes making up the single or double values such that the most significant bytes are grouped together, followed by the second most significant bytes, etc. This is likely done because the most significant bytes, which hold the exponent and top mantissa bits in IEEE 754 floating-point values, correlate more with each other than with the remaining bytes in the same value. This assumption is supported by the **LNVs1** component that follows, which computes the byte-granularity difference sequence and, therefore, exploits precisely this similarity between the bytes in the same position of consecutive values. The **LZa6** component compresses the resulting difference sequence. It uses $n = 6$ to avoid bad matches that result in zero counts being emitted, which expand rather than compress the data. The chosen high value of n indicates that bad matches are frequent, as is expected with relatively random datasets (cf. Table 1).

5.2. Compression Ratio

Table 2 shows the compression ratios on the 26 datasets for the investigated compressors. The highest compression ratio for each dataset is highlighted. The bottom row lists the geometric mean of the compression ratios over all datasets for each compressor.

There are only two algorithms that yield highest compression ratios on these datasets, Zstd and SPDP. Zstd compresses fifteen of the datasets best and SPDP the remaining eleven. On average, Zstd provides the highest compression ratio by a substantial margin. This is, to a large degree, due to its great performance on num_plasma, which it compresses by over an order of magnitude more than any of the other compressors. It appears that Zstd is able to capitalize on the very low fraction of unique values in this dataset (cf. Table 1).

SPDP outperforms the remaining compression algorithms on the majority of the datasets and also on average. Its geometric-mean compression ratio is over 30% higher than that of bzip2, the next best algorithm. We believe this is an impressive result given that SPDP does not include any bit-granularity coders whereas the other algorithms do.

The single-precision datasets are derived from their double-precision counterparts. Yet, only ten of the thirteen single-precision datasets are more compressible, most notably msg_sweep3d.sp, which is over 2.4 times as compressible as msg_sweep3d.dp. In the remaining cases, the double-precision versions are more compressible. For instance, obs_spitzer.dp is 1.8 times as compressible as obs_spitzer.sp. Evidently, the lower mantissa bits that are dropped when converting a double- to a single-precision value tend to be more random than the retained mantissa bits, but this is not always the case.

Overall, SPDP is surprisingly efficient, delivers a record compression ratio on 11 datasets, and outperforms the other algorithms except Zstd on 18 of the 26 datasets. These results highlight the potential of automatic compression-algorithm synthesis and answer our second research question. *A competitive algorithm can be created from components that do not process data at bit granularity such as Huffman or arithmetic coders*.

Table 2: Compression ratios

Dataset	Blosc level 1	Blosc level 9	bzip2 fast	bzip2 best	FastLZ fast	FastLZ best	LZ4 fast	LZ4 best	LZO	Snappy	SPDP level 1	SPDP level 9	Zstd level 1	Zstd level 22
msg_bt.dp	1.00	1.04	1.10	1.09	1.05	1.05	1.06	1.07	1.05	1.06	1.28	1.33	1.11	1.12
msg_lu.dp	1.00	1.00	1.02	1.02	0.98	0.98	1.00	1.00	1.00	1.00	1.20	1.26	1.06	1.05
msg_sp.dp	1.00	1.00	1.08	1.05	1.00	1.00	1.00	1.01	1.00	1.00	1.27	1.30	1.06	1.07
msg_sppm.dp	1.76	1.62	6.78	6.93	5.08	5.70	5.28	6.73	6.78	4.82	4.66	5.05	7.25	10.26
msg_sweep3d.dp	1.01	1.02	1.06	1.29	1.00	1.00	1.02	1.02	1.02	1.02	1.31	3.01	1.87	2.86
num_brain.dp	1.00	1.00	1.04	1.04	0.98	0.98	1.00	1.00	1.00	1.00	1.14	1.20	1.06	1.10
num_comet.dp	1.03	1.07	1.14	1.17	1.05	1.06	1.08	1.09	1.08	1.08	1.16	1.16	1.16	1.39
num_control.dp	1.01	1.01	1.03	1.03	0.99	0.99	1.01	1.01	1.02	1.01	1.02	1.01	1.06	1.06
num_plasma.dp	1.00	1.05	1.38	5.79	1.41	1.41	1.33	1.39	1.50	1.19	1.29	33.17	381.24	408.26
obs_error.dp	1.00	1.07	1.30	1.34	1.26	1.26	1.27	1.29	1.27	1.30	1.14	1.61	1.52	5.61
obs_info.dp	1.00	1.01	1.10	1.22	1.04	1.07	1.08	1.13	1.10	1.05	1.22	1.95	1.20	4.10
obs_spitzer.dp	1.00	1.02	1.29	1.75	1.04	1.05	1.05	1.20	1.14	1.06	1.00	0.98	1.18	3.27
obs_temp.dp	1.00	1.00	1.02	1.02	0.97	0.97	1.00	1.00	1.00	1.00	1.02	1.03	1.04	1.04
msg_bt.sp	1.00	1.04	1.13	1.13	1.06	1.06	1.06	1.07	1.08	1.06	1.43	1.48	1.14	1.22
msg_lu.sp	1.00	1.00	1.03	1.04	0.97	0.97	1.00	1.00	1.00	1.00	1.30	1.35	1.07	1.07
msg_sp.sp	1.00	1.00	1.16	1.14	1.00	1.00	1.00	1.01	1.08	1.00	1.47	1.53	1.11	1.26
msg_sppm.sp	3.18	1.69	8.14	8.74	7.85	8.37	8.55	8.73	8.63	6.26	6.93	7.69	9.84	13.52
msg_sweep3d.sp	1.01	1.02	1.10	2.35	1.00	1.00	1.02	1.03	1.03	1.02	1.45	7.32	5.52	6.07
num_brain.sp	1.00	1.00	1.09	1.11	0.98	0.98	1.00	1.00	1.00	1.00	1.26	1.35	1.13	1.13
num_comet.sp	1.03	1.07	1.11	1.12	1.06	1.06	1.08	1.09	1.09	1.08	1.18	1.18	1.15	1.15
num_control.sp	1.01	1.01	1.04	1.04	0.99	0.99	1.01	1.01	1.02	1.01	1.02	1.01	1.08	1.08
num_plasma.sp	1.00	3.93	1.53	8.65	1.12	1.97	1.00	1.09	1.22	1.02	3.26	34.74	254.98	369.22
obs_error.sp	1.00	1.02	1.28	1.34	1.17	1.18	1.12	1.20	1.25	1.18	1.28	2.03	1.31	5.63
obs_info.sp	1.00	1.01	1.12	1.33	1.02	1.07	1.07	1.13	1.13	1.05	1.60	2.28	1.21	3.62
obs_spitzer.sp	1.00	1.00	1.23	1.39	1.02	1.02	1.02	1.08	1.08	1.02	1.00	0.99	1.15	1.82
obs_temp.sp	1.00	1.00	1.04	1.05	0.97	0.97	1.00	1.00	1.00	1.00	1.01	1.00	1.08	1.08
GeoMean	1.07	1.11	1.31	1.59	1.20	1.24	1.21	1.25	1.26	1.19	1.42	2.09	2.22	3.09

5.3. Compression Speed

Table 3 lists the compression throughputs (in MB/s) on each dataset for the investigated compressors. The highest throughput for each dataset is highlighted. The bottom row shows the geometric mean throughput over all datasets for each compressor.

At level 1, both SPDP and Zstd compress at roughly 330 MB/s. At their best-compressing level, SPDP is much faster than Zstd. However, it should be noted that Zstd compresses better, on average, at its lowest level than SPDP at its highest level, so Zstd is clearly preferred. Among the other algorithms, all of which compress significantly less than SPDP, only Blosc level 1, LZ4 fast, and Snappy are faster than SPDP. The remaining algorithms are, on average, outperformed both in compression ratio and throughput by SPDP (and Zstd). Snappy is the fastest compressor, but it delivers one of the lowest compression ratios. Note that a high compression speed is of interest in scientific computing where large amounts of floating-point data are produced that may have to be compressed on the fly.

5.4. Decompression Speed

Table 4 lists the decompression throughputs (in MB/s) on each dataset for the investigated compressors. The highest throughput for each dataset is again highlighted. The bottom row shows the geometric mean throughput over all datasets for each compressor.

Zstd decompresses substantially faster than SPDP. It is 26% faster at level 1 and over two times faster in the best-compressing mode. Except for SPDP level 9, all algorithms decompress faster than they compress on average, in some cases by a large factor. For example, Zstd level 22 decompresses over 75 times faster than it compresses. In contrast, SPDP is a fairly symmetric algorithm that compresses and decompresses at about the same speed. A faster decompression throughput is useful in cases where a dataset is compressed once but decompressed multiple times, such as for datasets that are analyzed or visualized many times. In other cases, for example program checkpoints that are almost never read, the decompression speed is immaterial. Due to its symmetry, SPDP's decompression throughput is low and only higher than that of bzip2 and LZO. Again, Snappy is the fastest.

Table 3: Compression throughput in megabytes per second

Dataset	Blosc level 1	Blosc level 9	bzip2 fast	bzip2 best	FastLZ fast	FastLZ best	LZ4 fast	LZ4 best	LZO	Snappy	SPDP level 1	SPDP level 9	Zstd level 1	Zstd level 22
msg_bt.dp	747.4	194.2	8.8	8.4	146.2	233.4	1010.6	41.2	5.5	1011.7	413.1	199.2	310.1	2.2
msg_lu.dp	674.0	185.0	8.3	7.8	134.9	225.7	1406.4	38.0	5.6	1786.7	426.6	190.9	403.7	2.6
msg_sp.dp	746.1	187.9	8.4	8.0	144.4	227.6	1554.3	38.0	4.9	1636.8	387.6	200.5	443.7	2.1
msg_sppm.dp	819.3	355.9	6.7	5.7	520.6	472.1	900.4	52.6	6.3	1384.2	364.8	333.6	454.6	7.8
msg_sweep3d.dp	673.3	172.6	8.4	8.1	133.5	194.8	1226.3	38.8	5.7	1811.6	326.9	250.1	265.3	2.8
num_brain.dp	718.6	176.0	8.2	7.7	136.5	203.6	1249.0	38.0	5.5	2218.3	330.8	189.2	368.7	2.5
num_comet.dp	528.5	200.3	9.3	8.5	136.4	208.7	792.0	37.6	5.6	1946.9	298.8	186.3	365.5	2.5
num_control.dp	758.8	188.2	8.3	7.7	138.7	235.7	1824.1	38.0	5.7	2189.4	312.4	163.0	379.1	2.8
num_plasma.dp	677.5	194.7	9.7	2.8	157.1	304.6	295.9	46.2	7.8	477.3	388.9	277.7	1364.2	96.0
obs_error.dp	718.2	177.2	9.6	9.1	149.0	205.7	365.2	43.3	3.3	339.4	277.7	181.4	166.0	3.9
obs_info.dp	420.8	113.3	8.5	8.2	89.3	126.4	493.5	36.2	6.0	931.7	212.8	122.4	297.6	3.0
obs_spitzer.dp	719.3	176.5	9.4	9.6	128.4	170.7	548.9	39.4	4.1	445.2	371.5	155.4	148.0	1.5
obs_temp.dp	478.5	188.5	8.2	7.7	111.3	243.4	1051.0	38.2	5.6	1211.0	333.6	138.2	251.4	4.3
msg_bt.sp	745.5	185.3	9.2	8.7	139.7	217.0	965.3	40.1	5.4	814.6	379.1	208.9	214.2	2.4
msg_lu.sp	709.5	169.4	8.6	8.1	128.8	203.2	1171.1	37.4	5.6	1460.5	434.1	188.4	321.5	3.5
msg_sp.sp	679.9	179.4	9.0	8.4	137.8	206.2	1119.8	38.5	5.2	1602.9	376.7	207.9	333.5	2.5
msg_sppm.sp	730.0	318.3	5.6	4.8	539.9	385.9	1496.8	157.4	16.7	1784.0	458.6	376.5	691.0	15.6
msg_sweep3d.sp	512.7	169.2	8.7	4.0	127.6	174.4	1543.8	37.5	5.6	1335.4	350.0	217.5	638.3	15.0
num_brain.sp	717.4	183.3	8.6	7.9	120.3	174.4	1368.7	35.7	5.4	1559.6	262.0	187.1	315.6	3.7
num_comet.sp	474.1	202.1	9.1	8.3	128.0	204.6	814.2	37.1	5.6	1166.6	222.7	169.0	346.4	3.7
num_control.sp	673.1	190.9	8.3	7.7	130.0	260.3	1070.3	38.7	5.7	1701.6	289.7	167.4	334.9	3.8
num_plasma.sp	394.7	314.8	9.3	3.0	104.5	170.6	807.7	39.4	6.5	314.8	292.7	170.4	965.2	74.6
obs_error.sp	400.3	126.4	9.3	8.8	127.8	249.8	354.6	37.3	2.5	238.7	386.1	149.3	132.9	3.1
obs_info.sp	325.8	90.9	8.4	8.6	89.0	107.0	495.2	33.6	5.9	659.1	170.9	161.7	200.0	2.4
obs_spitzer.sp	521.4	147.8	9.1	8.8	118.4	148.9	430.2	35.6	3.7	351.1	301.1	169.9	179.7	1.7
obs_temp.sp	444.2	176.1	8.2	8.0	90.2	134.2	936.5	33.8	5.6	1348.2	258.7	132.1	270.7	5.1
GeoMean	597.7	183.6	8.5	7.2	140.3	208.1	870.3	40.8	5.5	1024.6	323.6	189.2	333.5	4.4

Table 4: Decompression throughput in megabytes per second

Dataset	Blosc level 1	Blosc level 9	bzip2 fast	bzip2 best	FastLZ fast	FastLZ best	LZ4 fast	LZ4 best	LZO	Snappy	SPDP level 1	SPDP level 9	Zstd level 1	Zstd level 22
msg_bt.dp	945.1	956.8	20.7	18.4	595.9	563.1	1854.4	1714.5	277.5	3470.8	430.4	203.7	478.6	269.3
msg_lu.dp	805.5	872.2	19.4	16.9	539.4	501.6	2081.0	2144.2	384.0	3624.9	453.9	199.3	422.6	289.6
msg_sp.dp	854.9	943.1	20.2	17.8	546.7	511.7	2149.9	1146.4	311.0	3741.9	412.6	198.8	469.2	231.7
msg_sppm.dp	960.8	1042.5	61.6	52.1	844.5	1002.4	1691.1	2450.3	327.8	3409.6	426.6	333.3	811.1	891.3
msg_sweep3d.dp	742.8	725.9	19.4	17.0	522.9	522.9	2070.6	1783.4	344.7	3313.6	431.5	122.2	393.7	323.9
num_brain.dp	795.1	734.8	18.8	16.5	649.4	495.9	2118.7	1831.6	384.0	3019.6	373.2	189.6	398.1	221.6
num_comet.dp	645.4	975.6	19.9	17.5	549.3	504.2	1634.4	1406.6	306.1	2529.7	315.0	188.3	381.8	232.0
num_control.dp	782.1	809.3	19.6	17.2	594.8	505.8	1398.4	2319.8	366.4	3127.1	360.1	158.3	408.5	424.1
num_plasma.dp	868.7	580.3	20.3	39.7	384.9	363.0	676.3	914.7	158.8	1119.0	437.2	84.0	1599.6	989.2
obs_error.dp	890.6	513.6	20.7	17.9	392.5	408.4	1605.5	1566.7	208.6	1857.6	298.7	122.1	316.9	457.7
obs_info.dp	592.2	442.0	18.4	16.4	298.6	267.9	1021.7	717.6	147.5	2313.9	306.5	123.6	237.3	301.1
obs_spitzer.dp	867.1	809.0	19.3	19.9	457.6	439.0	1075.3	1181.0	167.4	1966.8	377.1	150.9	404.5	289.8
obs_temp.dp	551.0	870.4	19.1	16.7	369.8	410.3	1162.2	1637.7	245.4	1871.0	340.4	165.9	310.2	187.4
msg_bt.sp	902.7	857.7	20.8	18.3	527.6	452.8	1733.6	979.8	213.2	2010.2	401.7	184.8	396.3	270.3
msg_lu.sp	714.1	915.4	19.5	17.2	454.6	427.6	1344.0	1784.3	326.4	2743.2	462.0	187.8	403.9	314.2
msg_sp.sp	749.6	728.6	20.5	18.2	439.4	443.6	2441.2	1663.2	165.0	2166.5	393.2	191.8	398.1	211.3
msg_sppm.sp	845.7	813.6	72.6	69.8	880.3	844.7	1504.3	2113.0	245.9	1672.0	461.5	312.3	1007.1	1017.1
msg_sweep3d.sp	911.1	663.9	19.0	25.1	638.4	363.7	1172.6	1732.2	187.0	2641.8	370.6	99.5	908.3	498.8
num_brain.sp	897.2	919.6	19.2	17.3	625.9	429.5	1871.9	1326.6	248.5	2198.7	342.2	157.6	323.7	202.8
num_comet.sp	840.0	805.8	21.0	18.1	409.5	430.6	1844.2	1613.9	253.1	2254.3	340.9	166.8	384.3	271.3
num_control.sp	821.8	912.7	19.4	16.9	705.3	460.8	1376.3	1443.0	338.7	3260.9	296.2	150.1	334.5	301.0
num_plasma.sp	462.5	568.0	22.1	39.0	275.4	342.0	1137.8	758.0	126.2	1289.7	406.0	105.6	2403.4	757.6
obs_error.sp	511.3	532.2	19.9	18.4	297.8	294.7	556.8	503.8	149.6	865.6	267.9	129.8	245.4	328.9
obs_info.sp	474.0	444.2	18.3	16.0	254.4	245.0	611.4	739.5	113.8	1152.7	125.1	134.1	206.5	244.3
obs_spitzer.sp	871.0	973.1	19.5	17.7	385.1	399.0	1058.4	1029.7	136.6	1506.3	390.4	154.8	392.8	197.3
obs_temp.sp	596.5	799.3	18.2	17.0	393.6	546.6	1252.2	2048.7	206.0	2447.8	286.4	136.4	281.2	203.6
GeoMean	749.1	756.2	21.6	20.7	476.5	447.4	1385.1	1379.4	228.7	2211.0	355.6	159.5	449.3	330.3

6. Summary and Future Work

In this paper, we describe the design, analyze the structure, and evaluate the performance of SPDP, an automatically synthesized lossless compression algorithm for single- and double-precision floating-point data. It is the best-compressing out of the 9,400,320 possible four-stage algorithms that can be built from our set of 48 algorithmic components that does not include any bit-level coders. SPDP yields the highest compression ratio on eleven of the 26 tested datasets and outperforms all of the evaluated compressors except Zstd. More importantly, our analysis represents a first step in a new direction aimed at improving our *understanding* of how to build effective domain-specific compression algorithms. First, by systematically generating candidates and analyzing the structure of the best resulting algorithm, we were able to gain insight into its operation and learned how to handle mixed-precision datasets. Second, we were able to demonstrate that a competitive algorithm can be created based solely on transformations that do not process data at bit granularity.

In future work, we would like to employ our approach in other domains to further improve our understanding of what makes an effective compression algorithm. To enhance the throughput, we intend to add optimized code generation that can interleave the operation of consecutive components and thus avoid repeated writing out and reading in of data between each pair of components. To boost the compression ratio, we could include bit-granularity components. Moreover, we want to study algorithms like Zstd and extract the key components from them so that we can synthesize even better algorithms.

Acknowledgments

The work reported in this paper was supported by the U.S. National Science Foundation under Grants 1217231 and 1438963 as well as a REP grant from Texas State University.

References

[1] Burtscher, M. and N.B. Sam. “Automatic Generation of High-Performance Trace Compressors.” *Int. Symposium on Code Generation and Optimization*, pp. 229-240. 2005.

[2] Burtscher, M., F. Hesaaraki, H. Mukka, and A. Yang. “Real-Time Synthesis of Compression Algorithms for Scientific Data.” *ACM/IEEE International Conference for High-Performance Computing, Networking, Storage and Analysis*, pp. 264-275. 2016.

[3] Coplin, J., A. Yang, A. Poppe, and M. Burtscher. “Increasing Telemetry Throughput Using Customized and Adaptive Data Compression.” *AIAA SPACE and Astronautics Forum and Exposition*. 2016.

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

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

[6] HDF5 compression filters: https://support.hdfgroup.org/services/contributions.html.

[7] HDF5 framework: https://support.hdfgroup.org/HDF5/.

[8] Hsu, W.H. and A.E. Zwarico. “Automatic Synthesis of Compression Techniques for Heterogeneous Files.” *Software: Practice and Experience*, 25(10):1097-1116. 1995.

[9] Kattan, A. and R. Poli. “Evolutionary Synthesis of Lossless Compression Algorithms with GP-zip3.” *IEEE Congress on Evolutionary Computation*, 1(8):18-23. 2010.

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

[11] Wu, M.S. and Y.L. Lin. “Genetic Algorithm with a Hybrid Select Mechanism for Fractal Image Compression.” *Digital Signal Processing*, 20(4):1150-1161. July 2010.

[12] Yang. A., H. Mukka, F. Hesaaraki, and M. Burtscher. “MPC: A Massively Parallel Compression Algorithm for Scientific Data.” *IEEE Cluster Conf.*, pp. 381-389. 2015.

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