CUSTOMIZED DATA COMPRESSION - AUTOMATICALLY SYNTHESIZING

EFFECTIVE DATA COMPRESSION AND DECOMPRESSION

ALGORITHMS

by

Hari Santhosh Manikanta Kumar Mukka, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
December 2014

Committee Members:

Martin Burtscher, Chair

Anne Ngu

Dan Tamir

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

I would like to dedicate this thesis to my family.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Dr. Martin Burtscher, for his continuous support throughout my research. I am glad that I worked under a professor who understood me in my odd times and whose guidance helped me in all the time of research and writing this thesis.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Anne Ngu and Dr. Dan Tamir for their encouragement and support.

I would like to thank my parents for encouraging me to study abroad and helping me in all the possible ways they can. Last but not least, I thank my uncle, without whose support I would not have started my thesis.

This project was funded in part by the U.S. National Science Foundation under Grant No. 1217231.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

With the exponential increase in the amount of data humans are generating, there is a progressive need for developing effective high-speed data compression techniques. However, developing an algorithm that just compresses data well is not sufficient. A good compression technique should also minimize the compression and decompression time. This project focuses on automatically synthesizing compression and decompression algorithms for a given data type. In particular, it employs search techniques to mix and match different transformation, prediction, and encoding components to determine the most efficient configurations.

CHAPTER 1

INTRODUCTION

1.1 Data Compression

Data compression is reducing the size of data for faster data transfer and to store data using less storage space. It is ubiquitous. For example, most of the images on web pages are compressed, typically using JPEG or GIF, modems and fax machines use compression, HDTV uses MPEG-2 for compression, and several file systems automatically compress files when they are stored.

1.1.1 Compression Techniques

A compressor is usually a combination of two algorithms, one that takes an input “X” and generates “Xc” that requires fewer bits and a reconstruction algorithm that operates on the compressed representation to generate “Y”, which may or may not be the same as the original data. Based on this reconstruction requirement, compression techniques are divided into lossless and lossy compression algorithms.

Lossless Algorithms

These algorithms can reconstruct the original data exactly from the compressed data. For example, lossless compression is required for program executables where every bit matters. It is employed in winzip, gzip, and other compression utilities. There are many situations where we require the decompressed file to be same as the original file. In situations where this requirement is not necessary, we can use lossy compression.

Entropy

Entropy is a measure of the average number of bits required to encode each symbol in the output of the source. The best that lossless compression can do (in the absence of a data model) is to encode the output of a source with an average number of bits equal to the entropy of the source.

Lossy Algorithms

These algorithms typically compress better but can only approximate the original data. In many applications this approximation is not a problem. For example, when storing or transmitting speech, the exact value of each sample of speech is not necessary similarly for video and image compression, data can be reconstructed depending on the quality required.

1.1.2 Measures of Performance

Once an algorithm is developed, we need to be able to measure its performance. Because of different areas of application, different terms have been developed to describe and measure the performance of compression algorithms. We could measure the relative complexity of the algorithm, the memory required to implement the algorithm, how fast the algorithm performs on a given machine, the amount of compression, and how close the reconstruction resembles the original. In this thesis, I have used the amount of compression achieved and the algorithm speed to measure the performance.

1.1.3 Modeling and Coding

At a high level, most data compression algorithms comprise two stages, a data model and a data encoder. Roughly speaking, the goal of the model is to accurately predict the data. The residual between this prediction and the actual data is then compressed with the encoder. The encoder maps the input data to bit sequences in such a way that frequently encountered values will produce shorter output than infrequently encountered data. This project's objective is to develop a suite of different generic data models that can be chained to create more sophisticated models.

For example, value predictors are a good source for such data models. Various value predictors exist that "guess" the next value in a sequence. The difference between the actual value and the predicted value will be close to zero if the model is good for the given data. As the resulting sequence of differences is simply another sequence of values, the output of one predictor can be used as input to another predictor and so forth. The output of the last predictor is then encoded. Search techniques decide the best combinations and permutations of predictors to be used to generate an effective compression technique for a given data set.

Each component will have a corresponding inverse component. For instance, for each value predictor there exists an inverse predictor that takes the residual sequence as input and regenerates the original sequence of values. This ensures that, for any synthesized combination of predictors, there not only exists a decompression algorithm but it can, in fact, be synthesized as well. In other words, the de-compressor can automatically be generated based on the chosen compression configuration.

Figure 1.1 represents an overview of the thesis idea; components A, B, and C are chained to compress an input file and then the complements of each component are used in reverse order to decompress the file.

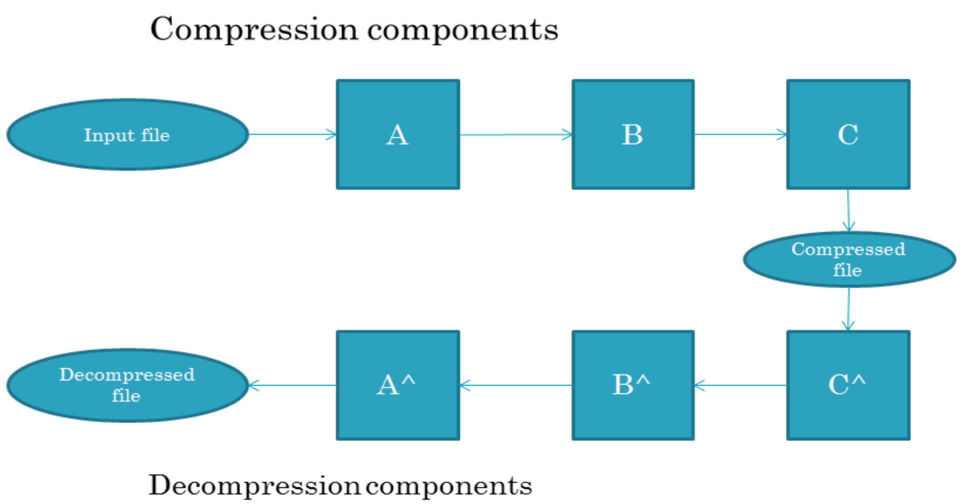

Figure 1.1 Overview of the approach

1.2 Contributions

- I combined different algorithmic components to form novel, effective compression algorithms.
- I used exhaustive search to determine the best possible solutions within the given search domain.
- I used a genetic algorithm to find good solutions quickly.
- This approach is flexible and can easily be extended to include more, fewer, or different components to improve the performance.

The rest of the thesis is organized as follows. Chapter 2 summarizes related work. Chapter 3 describes the different components used in the thesis. Chapter 4 presents the evaluation methods. Chapter 5 discusses the results. Chapter 6 provides a summary and conclusions.

CHAPTER 2

BACKGROUND

2.1 Huffman Coding

The Huffman code is an optimal prefix code and the process of using a Huffman code is called Huffman coding (Huffman 1952). It is a common technique employed in entropy encoding and in lossless compression. In this technique, the values that occur more frequently have shorter code words than less frequent values. Huffman coding works by creating a binary tree of nodes. The simplest construction algorithm uses a priority queue where the value with the lowest probability is given the highest priority. The procedure for building this tree is:

1. Start with a list of free nodes, where each node corresponds to a symbol in the alphabet.
2. Select the two free nodes with the lowest weight from the list.
3. Create a parent node for these two nodes selected where the weight is equal to the sum of the weights of the two child nodes.
4. Remove the two child nodes from the list and insert the parent node into the list of free nodes.
5. Repeat the process starting from step 2 until only a single tree remains.

After building the tree, the prefix code for each symbol is created by traversing the tree from the root to the corresponding leaf node. It assigns 0 for a left branch and 1 for a right branch.

In Adaptive Huffman coding, the Huffman tree includes a counter for each symbol and is updated each time the corresponding symbol is encoded. This algorithm generates codes that are much more effective than static Huffman coding. This algorithm requires only one pass over the input and adds little or no overhead to the output. This version is slower than static Huffman coding as the tree is potentially rebuilt for each symbol.

2.2 Arithmetic Coding

Arithmetic coding (Langdon 1984) is a form of entropy coding. It is similar to Huffman coding except that it encodes the entire message in a single number, a fraction n where $(0.0 < n < 1.0)$.

Arithmetic coding generates a unique identifier or tag to encode the sequence. In the first phase, a unique identifier or tag is generated for a given sequence of symbols. This tag is then given a binary code. Thus, a unique arithmetic code can be generated for a sequence of length m without the need of generating code words for all sequences of length m as in Huffman coding.

2.3 Run-Length Coding

This model is derived from the Capon model, a two state Markov model with states S_w and S_b (S_w corresponds to the case where the pixel that has just been encoded is a white pixel and S_b corresponds to the case where the pixel that has just been encoded is a black pixel.) The transition probabilities ($P(w/b)$ and $P(b/w)$) and the probability of being in each state ($P(S_w)$ and $P(S_b)$) completely specify this model. The main idea of this model is that a pixel of a particular color is likely to be followed by the same color pixels. So, it is better to code the repeated length rather than coding each pixel. For example, if there are 190 white pixels followed by 200 black pixels we can simply code them as the two values “190, 200” instead of coding each pixel.

2.4 Dictionary-Based Compression

2.4.1 LZ77

Jacob Ziv and Abraham Lempel presented a dictionary-based compression algorithm in 1977 (Lempel and Ziv 1977). It works on the fact that words and phrases within a text file are likely to be repeated. A pointer is used to point to the previous occurrence of the current element and a number specifies the number of elements to be matched. The dictionary is simply a portion of the previously sequence of elements. A sliding window is used to encode the sequence, which consists of a search buffer that contains the portion of the previously encoded sequence and a look-ahead buffer that contains the portion of the sequence to be encoded. The algorithm searches for the longest match and outputs a triplet <o, l, c>, where *o* is the offset of the match, *l* is the length of the match, and *c* is the next element to be encoded. Figure 2.1 presents the LZ77 algorithm.

```
While (lookAheadBuffer not empty) {
get a reference (position, length) to longest match;
if (length > 0)
{
 output (position, length, next symbol);
 shift the window length+1 positions along;
  }
else {
output (0, 0, first symbol in the lookahead buffer);
shift the window 1 character along;
  }
}
```

Figure 2.1 LZ77 algorithm

2.4.2 LZ78

Ziv and Lempel presented another dictionary-based compression scheme in 1978 (Ziv and Lempel 1978), which maintains an explicit dictionary. This dictionary has to be built both at the encoding and the decoding side and must follow common rules. A code word in this algorithm consists of two elements <i, c>, where *i* is the index referring to the longest non-matching literal and *c* is the first non-matching symbol. In addition to outputting each code word, it is also added to the dictionary. When a symbol that is not yet in the dictionary is encountered, then a code word with index value 0 is added to the dictionary. The only drawback of this version is that the dictionary keeps on growing, which is limited in the later versions. Figure 2.2 presents the LZ78 algorithm.

```
w := NIL;
while ( there is input ) {
  K := next symbol from input;
  if (wK exists in the dictionary) {
     w := wK;
  } else {
   output (index(w), K);
   add wK to the dictionary;
   w := NIL;
  }
}
```

Figure 2.2 LZ78 algorithm

2.5 Integer Compression Algorithms

2.5.1 Golomb Codes

This compression technique belongs to the family of codes designed for integer compression. It works on the assumption that the larger the integer, the lower its probability of occurrence. The simplest code for this situation is a unary code. The unary code for a positive integer *n* is *n* 1s followed by 0. For example, the code for 3 is 1110.

For each input value, the Golomb code outputs a code word that is a combination of a quotient code and a remainder code of that value. The quotient code is the unary encoding of the quotient obtained by dividing the input value by some fixed value and the remainder code is the binary encoding of the remainder.

2.5.2 Elias Gamma Coding

This coding algorithm is used for integers whose upper bound cannot be determined beforehand. In this technique, each binary representation of the input is prepended with zeros whose count is equal to the difference of the number of bits required to represent the value and 1. The Elias gamma code of 7 is 00111.

2.5.3 Fibonacci Coding

The Fibonacci code is a universal code that encodes positive integers into binary code words. Fibonacci coding works as follows.

1. The input value *N* is subtracted from the largest Fibonacci number equal to or less than *N*.
2. If the result is the i^{th} Fibonacci number, then 1 is placed at position i-2 in the code word.
3. These steps are repeated until the result is zero.
4. Finally, an additional 1 is placed after the rightmost digit in the code word.

CHAPTER 3

RELATED WORK

Ahmed Kattan and Riccardo Poli proposed a system called GP-zip3 (Kattan and Poli 2010) that uses genetic programming to find optimal ways to combine standard compression algorithms. GP-zip3 evolves programs with multiple components. One component divides the data into blocks. These blocks are then projected onto a two-dimensional Euclidean space via two further (evolved) program components. Similar data blocks are grouped using the K-means clustering algorithm. Each cluster is then labeled with the optimal compression algorithm for its member blocks. Once a program that achieves good compression has been evolved, it can be used without further evolution.

Automatic synthesis of compression technique for heterogeneous files (Hsu and Zwarico 1995) is presented by William H. Hsu and Amy E. Zwarico in 1995. Each block of data is compressed using a different algorithm, which is determined using a statistical method. The actual compression is accomplished in two phases. The first phase determines the compressibility of each block using some quantitative metrics. A block of data is considered to be fully compressed if the metrics fall below a certain threshold value. The compression shifts to the next block when the threshold value is reached. In the second phase, adjacent blocks that use the same compression algorithm are grouped together for better performance. A compression history, required for decompression, is automatically generated in this phase.

Wenbin Fang, Bingsheng He, and Qiong Luo presented "Database Compression on Graphics Processors" (Fang, He and Luo 2010) to overcome the data transfer overhead, which is an important factor for query co-processing performance on GPU. Their approach uses a compression planner along with a cost model to find an optimal combination among nine different compression schemes. The compression planner is a combination of a tactical planner and a strategic planner. The tactical planner uses a rule-based method to automatically prune the search space for a predefined maximum number of schemes, and the strategic planner allows the developers to specify their goals. The cost model estimates the execution time based on the parallel execution mechanism of CUDA-based GPUs.

Burtscher Martin and Sam Nana B presented TCgen (Burtscher and Sam 2006), a trace compression tool that automatically generates portable, customized, high-performance trace compressors. The user provides a description of the trace format and selects one or more predictors for compression. TCgen then translates this description into C source code and optimizes it for the specified trace format and predictors.

Suman K. Mitra, Murthy C. A, and Malay K. Kundu proposed a methodology for compressing fractal images using a genetic algorithm (Mitra, A and Kundu 1998). Initially, fractal codes (F_i) are computed for each domain block (D_k). 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 if it is above the threshold. The main aim of this classification is to obtain higher compression and to reduce the encoding time. The final step uses a genetic algorithm to find a good match (optimal solution) to the rough domain blocks.

“Automatic generation of parallel sorting algorithms” (Garber, et al. 2008) by Brian A. Garber, Dan Hoeflinger, Xiaoming Li, Maria Jesus Garzaran, and David Padua discusses a library generator that examines the input characteristics for selecting the best sorting algorithm. The sorting routine uses a training phase, in which an empirical search is employed to determine the values of the parameters on the target machine, and a runtime phase, which examines the input for certain characteristics and selects the appropriate sorting routine.

The Fastest Fourier Transform in the West (FFTW) (Frigo and Johnson 1997) is a free software library for computing discrete Fourier transform (DFT). FFTW uses a planner to maximize the performance, whose input is a problem and a loop of DFTs. The planner measures the actual runtime of many different plans and selects the fastest one. Plans are generated according to rules that recursively decompose problems into smaller sub problems.

Except for “Database compression on graphic processors”, the papers described above either do not target data compression at all or only consider complete compression algorithms as components. Moreover, they all use an imprecise method to select components. In contrast, this thesis 1) uses an exhaustive search to determine the truly best solution with small numbers of components (in addition to a genetic algorithm for larger numbers of components), 2) presents a general approach that works in any domain (though we evaluate it on floating-point data because floating-point data are both widely used and hard to compress losslessly), 3) considers a much larger number of components than the related work, and 4) does not only study combinations of existing compression algorithms but also synthesizes brand new algorithms by combining parts of algorithms in ways that have never before been tried.

CHAPTER 4

COMPONENTS

This section describes the various components that are used to synthesize the compression algorithms. Each component has a corresponding inverse component that performs the opposite action, which is needed to synthesize the decompression algorithm.

4.1 Mutators

Mutators simply change bit value(s) of an element.

INV (Inverse)

This component flips all the bits in every element.

NEG (Negation)

This component negates each element.

MSB (Most Significant Bit)

If the most significant bit of an element is 1, then all the remaining bits are flipped.

4.2 Predictors

Predictors predict (or extrapolate) the next value in a sequence based on previous values. The predicted value is then subtracted from or XORed with the true value. This operation results in many zero bits if the prediction is accurate.

LNV*n* (Last n Value)

This component divides the input into chunks of 1024 elements and, in each chunk, each element is predicted using the n^{th} previous element, starting from $(n+1)^{st}$ element.

PLY*n* (Polynomial)

This component works by fitting an order *n* polynomial through the previous values and uses the resulting polynomial to extrapolate the next value.

SEL*n* (Select)

The SEL component predicts the elements using the most recent element that has the same n^{th} byte value as the present element.

FCM*n* (Finite-Context-Method Predictor)

The FCM predictor contains two tables. A hash value computed using the *n* most recently encountered values is stored in the predictor's first-level table. The number of values per line, i.e., *n*, determines the order of the predictor. The hash is then used to index the predictor's second-level table. During predictions, a hash table lookup is performed in the hope that the next value will be equal to the value that followed last time the same sequence of *n* previous values (i.e., the same hash) was encountered. Thus FCM*n* predictor can memorize long arbitrary sequences of values and accurately predict them when they repeat.

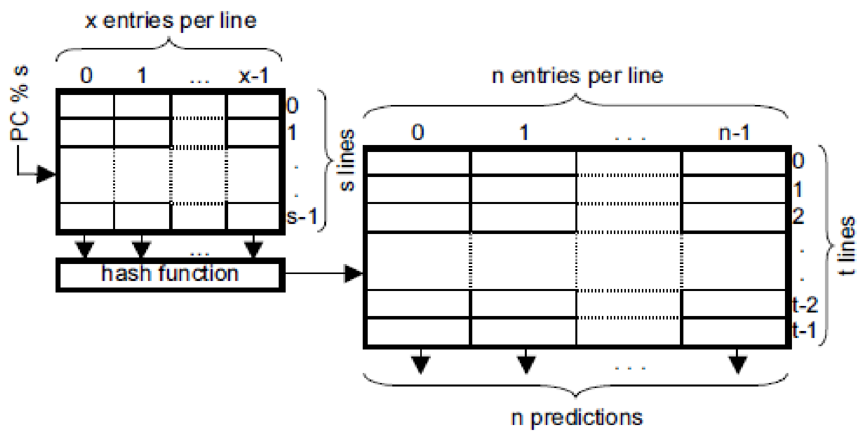

Figure 4.1 FCM predictor

DFCM*n* (Differential Finite Context Method Predictor)

The differential finite context method predictor is similar to FCM predictor except that it predicts and is updated with the differences (strides) between consecutive trace entries rather than original values. To form the final prediction, the predicted stride is added to the most recently seen value. DFCM predictors often make better use of hash tables and, unlike FCM predictors, can predict values that have never been seen before.

4.3 Reducers

Reducers are the components that perform actual compression to reduce the length of the sequence. They attempt to replace sequences of values with different sequences that are shorter.

ZE (Zero Eliminator)

This component divides data into chunks of size equal to the number of bits in the input data type. For each chunk, it emits a bitmap indicating whether the corresponding element was a zero or not. This bitmap is followed by all non-zero values.

RLE (Run Length Encoding)

In this compressor, repetitions of the same value are replaced with a single datum and a count. For example, consider a single scan line of a screen containing black text on white background, with B representing black pixel and W representing white pixel.

WWWWBBBBBBWWWBBBWWWWWWWWWWWWWBBWWWWWW.

If run-length-encoding is applied to the above line, we obtain the following output.

4W6B3W3B13W2B6W.

After this compression, the number of characters stored is 15 rather than 37.

RLEa

One problem of the RLE component above is that it greatly expands the output if there are no repeating values. This happens because every value is preceded with a count of one. To reduce the number of one counts, RLEa alternately records a repeating count plus the repeated value followed by a non-repeating count plus all the non-repeating values. In RLEa, both the repetition count and the count of the number of non-repeating elements are stored together in a single word using half the bits each.

RLEb

The only difference between RLEa and RLEb is that both counts are stored in separate words, thus requiring more space for storing counts but also extending the range of representable counts.

LZB*n* (Lempel Ziv Burtscher)

This component is a variation of the well-known and widely used Lempel Ziv algorithm (Shanmugasundaram and Lourdusamy 2011). It works on the principle that patterns of values are likely to be repeated. Most of the Lempel Ziv algorithms output a triple <o, l , c>, where o is the offset of the match, l is the length of the match and c is the next symbol of the match. Other versions output two elements <i, c>, where i is the index of the longest matching pattern and c is the first non-matching literal. LZB does not use any of the above mentioned methods for matching instead uses a hash table to figure out where a previous match might be, but only considers the match if the first *n* values do, in fact, match.

4.4 Shufflers

Shufflers are used to change the order of values or bits from their original position to some other position.

SWP

This component reverses the endianess of every other element.

DIM

This component reorders the elements based on the given dimensionality. For example, with a dimension of three, then the elements at positions 0, 3, 6, … are copied to the output array followed by elements at 1, 4, 7, ... followed by elements at 2, 5, 8, …. This puts elements from the same dimension next to each other.

BIT

This component divides data into chunks of size equal to the number of bits of the selected data type. From each chunk, the first bit values of all the elements are stored in the first element of the output sequence, the second bit values to the second element and so on till the last bit values of all the elements of the chunk are processed. This puts the n^{th} bits of each element next to each other.

4.5 Expanders

These components increase the length of the output in the hope that, by doing so, they expose patterns that allow the following components to compress better. The two expanders discussed below double the size of input sequence.

hPLY (hybrid Polynomial)

This component records which of the several PLY components gave the best prediction as well as the sign bit of the difference between the predicted and the actual value. This value is followed by the difference between the predicted and the actual value, i.e., every original value is converted into two values.

hLNV (hybrid Last n Value Predictor)

This component is similar to hPLY except that this component uses multiple last *n* value predictors instead of PLY predictors.

4.6 Search for Effective Algorithms

Two approaches are used to automatically determine the most effective compression algorithms.

4.6.1 Exhaustive Search

Exhaustive search generates all possible combinations of components using a given number of chained components and output the best solution. Once the optimal solution within the search space has been determined, the compression ratio and the runtime of the found algorithm are presented in the output. Since the search time is exponential in the number of chained components, this approach is only tractable for short chains of components.

4.6.2 Genetic Algorithm

A genetic algorithm (GA) (Genetic Algorithm n.d.) is a heuristic search algorithm that is based on the evolutionary ideas of natural selection and genetics. Heuristics are often used for generating useful solutions to optimization and search problems. Genetic algorithms employ techniques like crossover and mutation in an iterative process that comprises four steps.

Initialization of Genetic Algorithm

In the initialization phase, many individual solutions are generated randomly to form an initial population. The population size depends on the nature of problem.

Selection of Solutions

Each individual solution is evaluated using a provided fitness function. In my case, the fitness is simply the compression ratio. A portion of the existing population is then selected based on their fitness to “breed” a new generation using different genetic operators.

Genetic Operators

Every next generation of solutions is generated using genetic operators like crossover (recombination) and mutation (random changes) applied to the previous generation’s solutions. In crossover, two parents are selected with a probability that is proportional to their fitness. Then some of the components from one parent are combined with the remaining component of the other parent. In mutation, a single parent is chosen and one or a few of its components are randomly changed. This process continues until a new population of solutions of the desired size has been generated. The new generation thus produced will share many of the characteristics from the previous generation. As the best solutions are selected to generate the new population, the resulting new generation will likely also yield good results.

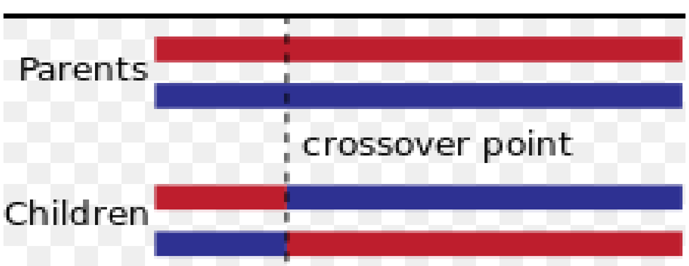

Figure 4.2 Crossover

Termination

The genetic algorithm terminates in the following cases.

1) After a fixed number of generations.

2) When a solution is found that meets a minimum fitness requirement.

3) When sufficiently many successive iterations do not produce better results.

CHAPTER 5

EVALUATION METHODS

5.1 System and Compiler

I have used a 64-bit system with 3.4 GHz Intel Xeon X5690 24 CPU, which has 6 cores, 64 kB L1 cache, 256 kB unified L2 cache, 12 MB L3 cache and 24 GB of main memory. The operating system is Red Hat Enterprise Linux and the compiler is gcc version 4.4.7. I used "-march=native -O3" compiler flags for each compressor.

5.2 Performance Metrics

Three different performance metrics are used in the thesis for evaluating the quality of the compression algorithms. They are the compression ratio, the decompression speed, and the compression speed. They are all higher-is-better metrics. They are defined as follows.

$$compression\ ratio = \frac{uncompressed\ size}{compressed\ size}$$

$$decompression\ throughput = \frac{uncompressed\ size}{decompression\ time}$$

$$compression\ throughput = \frac{uncompressed\ size}{compression\ time}$$

Note that the compression ratio has no unit while the decompression and compression speeds are throughputs measured in bytes per second.

5.2.1 Timing Measurement

All timing measurements in this thesis refer to the sum of the user and system time reported by the UNIX shell command time. In other words, the idle time such as waiting for disk operations is ignored.

5.3 Compression Algorithms

This section describes the different compression algorithms used in the thesis to compare the compression ratios.

BZIP2

BZIP2 (bzip2 2006) is a lossless, general-purpose file compression algorithm that operates at byte granularity. It implements the block sorting algorithm described by Burrows and Wheeler. It compresses data in blocks, where the block size is adjustable. It uses the Burrows-Wheeler transform to convert frequently occurring character sequences into strings and then uses a move-to-front transform and Huffman coding for compressing the data.

GZIP (GNU zip)

GZIP (The gzip home page 2006) implements a variant of the LZ77 algorithm and operates at byte granularity. It looks for repeating strings of length not greater than 256 bytes within a 32kB sliding window. It uses two Huffman trees, the first tree compresses the distances in the sliding window and the second one is used to compress the length of strings. The second tree is also used to compress individual bytes that were not part of any sequence. Duplicated strings are found using chained hash tables whose maximum length is determined by the command line argument.

FSD

The FSD compressor (Engelson, Fritzson and Fritzson 2000) implements fixed step delta algorithm proposed by Engelson, which iteratively generates difference sequences as it reads in a stream of doubles. The order determines the number of iterations. A zero suppress algorithm is then used to encode the final difference sequence, where each value is expected to have many leading zeros. Rapidly changing data compress better with lower orders whereas gradually changing data tend to benefit from higher difference orders.

PLMI

Lindstorm and Isenberg proposed the PLMI (Lindstorm and Isenburg 2006)compression, which implements a Lorenzo predictor for predicting 2D and 3D geometry data and a delta predictor for linear data. The delta predictor processes data similar to first order FSD algorithm. The predicted and true floating-point values are mapped to unsigned integers from which a residual is computed by a difference process. In the final step, the residual is encoded based on Schindler’s quasi-static probability model.

FPC

FPC (Martin and Ratanaworabhan, FPC: A High-Speed Compressor for Double-Precision Floating-Point Data 2008) is a lossless compression algorithm for linear sequences of double-precision floating point values. FCM and DFCM value predictors are used for predicting new values. The more accurate value from the two predictions is selected and is xored with the true value. As the XOR operation turns identical bits to zeroes, the result will have many leading zeroes if the predicted value is close to the true value. FPC encode the leading zeroes into a three bit code and is concatenated to a single bit that specifies the predictor used. The output contains the four bit code followed by the non-zero residual bytes. During decompression, FPC starts by reading the four-bit code, decodes the three-bit field and reads the specified number of residual bytes. The value is extended to 64-bits by adding zeroes to it. The resultant number is xored with either the FCM or DFCM prediction based on the one-bit field to recreate the original value.

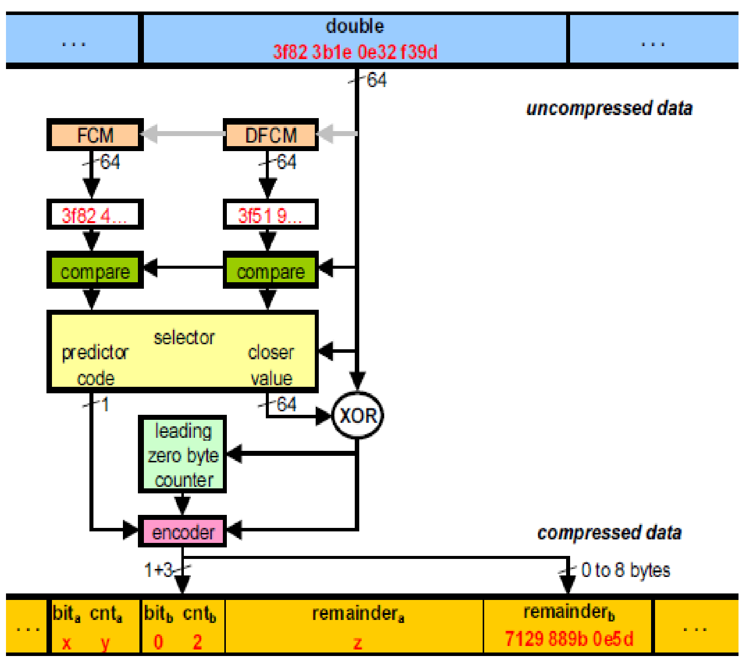

Figure 5.1 FPC compression algorithm

pFPC (parallel FPC)

pFPC (Martin and Paruj, pFPC: A Parallel Compressor for Floating-Point Data 2009) is a parallel implementation of the lossless FPC compression algorithm. In this approach, data is divided into chunks and multiple instances of FPC compress or decompress these chunks in parallel. The number of threads and the size of the chunk are selected by the user. The chunk size determines the number of consecutive doubles that make a full size chunk and the thread size determines the number of instances of FPC that work together. Chunks are assigned in a round-robin fashion to the threads.

gFPC (genetic FPC)

gFPC (Martin and Paruj, gFPC: A Self-Tuning Compression Algorithm 2010) is based on FPC and uses a genetic self-tuning approach. Each block of data is compressed multiple times using different hash function configurations. The number of configurations tested is called population size. Initial population can be random or it can be seeded with, for example, the FPC configuration. The fitness function determines the quality of each configuration based on the compression ratio. The configuration with highest fitness is written to the compressed output along with the compressed block of data. The next block is compressed with a new generation of configurations, which are produced using the best configurations from the previous generation. The smaller the block size, the more often a new generation is produced.

LZOP

LZOP (lzop n.d.) is a lossless data compression technique that focuses on compression and decompression speed rather than compression ratio. It is similar to gzip and uses LZO1X from the LZO (Lempel Ziv Oberhumer) [28] data compression library for compressing files. It is a block compression algorithm that compresses a block of data into matches (a sliding dictionary) and runs of non-matching literals. LZO produces good results for redundant data and, for better performance, overlapping and in place compression can be used. Decompression is performed in the reverse order. LZO1X is five times faster than gzip's Deflate algorithm.

5.4 Datasets Used

I used thirteen datasets from various scientific domains for evaluation. Each dataset consists of a one-dimensional binary sequence of IEEE 754 double-precision floating-point numbers and belongs to one of the following categories.

Observational Data

These datasets comprise measurements from scientific instruments.

- obs_error: data values specifying brightness temperature errors of a weather satellite
- obs_info: latitude and longitude of the observation points of a weather satellite
- obs_spitzer: data from the Spitzer Space Telescope showing a slight darkening as an extra-solar planet disappears behinds its star

- obs_temp: data from a weather satellite denoting how much the observed temperature differs from the actual contiguous analysis temperature field

Numeric Simulations

These datasets are the results of numeric simulations.

- num_brain: simulation of the velocity field of a human brain during a head impact
- num_comet: simulation of the comet Shoemaker-Levy 9 entering Jupiter's atmosphere
- num_control: control vector output between two minimization steps in weather satellite data assimilation
- num_plasma: simulated plasma temperature of a wire array z-pinch experiment

Parallel Messages

These datasets capture the messages sent by a node in a parallel system running NAS Parallel Benchmark (NPB) (Bailey, et al. 1995) and ASCI Purple (2006) applications.

- msg_bt: NPB computational fluid dynamics pseudo-application bt
- msg_lu: NPB computational fluid dynamics pseudo-application lu
- msg_sp: NPB computational fluid dynamics pseudo-application sp
- msg_sppm: ASCI Purple solver sppm
- msg_sweep3d: ASCI Purple solver sweep3d

Table 5.4.1 Statistical information about each dataset

Dataset	Size (megabytes)	Doubles (millions)	Unique values (percent)	1st order entropy (bits)	Randomness (percent)
msg_bt	254	33.30	92.9	23.67	94.7
msg_lu	185.1	24.26	99.2	24.47	99.7
msg_sp	276.7	36.26	98.9	25.03	99.7
msg_sppm	266.1	34.87	10.2	11.24	44.9
msg_sweed3d	119.9	15.72	89.8	23.41	97.9
num_brain	135.3	17.73	94.9	23.97	99.5
num_comet	102.4	13.42	88.9	22.04	93.1
num_control	152.1	19.94	98.5	24.14	99.6
num_plasma	33.5	4.39	0.3	13.65	61.9
obs_error	59.3	7.77	18.01	17.80	77.8
obs_info	18.1	2.37	23.9	18.07	85.3
obs_spitzer	189.0	24.77	5.7	17.36	70.7
obs_temp	38.1	4.99	100.0	22.25	100.0

The size of each dataset in megabytes and in millions of double-precision floating point values are listed in the first and second column, respectively. The third column represents the percentage of values that are unique in each dataset. The fourth column displays the first-order entropy of the values in bits. The last column displays the randomness of the datasets in percent, that is, it reflects how close the first-order entropy is to that of a truly random dataset with the same number of unique values.

CHAPTER 6

RESULTS

This section compares the results of the genetic approach with other compressors from the literature and compares the exhaustive search with the genetic algorithm. The population size used is 2420 in all cases and the number of generations is 10. I generated results for between 1 and 10 stages (i.e., the length of the component chain) and the best result for each dataset is presented in this section. All thirteen datasets described in Section 4.4 are used in the evaluation. The best result for each dataset is highlighted in each table.

6.1 Compression Ratio

Table 6.1.1 presents the best found compression ratios for up to ten stages when using the genetic approach.

Table 6.1.1 Compression ratio for stages 1 to 10 using genetic

Trace	Compression ratios for corresponding stages									
	1	2	3	4	5	6	7	8	9	10
msg_bt	1.062	1.206	1.229	1.229	1.232	**1.315**	1.207	1.244	1.243	1.255
msg_lu	1.000	1.170	1.255	1.523	1.544	1.268	1.563	**1.546**	1.307	1.298
msg_sp	1.000	1.219	1.255	1.268	**1.339**	1.295	1.309	1.310	1.326	1.237
msg_sppm	4.141	5.179	5.544	5.563	6.049	6.137	6.149	6.141	6.166	**6.274**
msg_sweep3d	1.017	1.215	1.292	1.318	1.352	1.356	1.351	1.355	1.364	**1.376**
num_brain	1.132	1.158	1.214	1.220	**1.235**	1.217	1.234	1.222	1.232	1.232
num_comet	1.081	1.254	1.307	1.339	1.348	1.342	1.354	1.307	**1.360**	1.350
num_control	1.013	1.096	1.128	1.128	1.137	1.132	1.129	1.128	**1.137**	1.126
num_plasma	1.063	1.281	2.258	2.307	2.425	2.749	2.825	1.337	1.583	**3.031**
obs_error	1.225	1.289	1.338	1.350	1.384	1.487	1.397	1.527	**1.561**	1.523
obs_info	1.006	1.200	1.252	1.278	1.262	1.256	1.282	1.286	**1.296**	1.285
obs_spitzer	1.039	1.216	1.251	1.258	1.266	1.276	1.266	1.266	**1.279**	1.274
obs_temp	1.062	1.095	1.120	1.120	1.123	1.120	**1.123**	1.120	1.118	1.120

Nine of the thirteen datasets got the best compression ratio using more than seven stages. No dataset achieved its best compression ratio with an algorithm that comprises fewer than five stages. Expectedly, there is a gradual increase in compression ratio with larger numbers of stages on most datasets. Initially, the increase is larger but quickly starts to plateau off. For some datasets, it even drops off after peaking at five or six stages. The best compression ratio is used in the rest of the evaluations.

6.2 Comparison

This section compares the compression ratios of my approach with the algorithms described in Section 5.3. The “custom” column represents the results of my approach.

Table 6.2.1 Compression ratio comparison

Trace	Compression ratio									
	BZIP2	GZIP	LZOP	FPC	pFPC	gFPC	DFCM	FSD	PLMI	Custom
msg_bt	1.100	1.120	1.045	1.28	1.18	1.300	**1.361**	1.074	1.245	1.315
msg_lu	1.017	1.045	0.995	1.171	1.095	1.201	1.249	1.000	1.196	**1.546**
msg_sp	1.072	1.1068	0.998	1.257	1.167	1.275	1.256	0.997	1.198	**1.339**
msg_sppm	**6.675**	6.200	4.927	5.235	3.985	4.850	4.231	2.354	5.029	6.274
msg_sweep3d	1.061	1.081	1.016	**2.850**	1.212	1.250	1.565	1.212	1.215	1.576
num_brain	1.032	1.057	0.994	1.156	1.118	1.156	1.232	1.100	1.124	**1.235**
num_comet	1.137	1.15	1.066	1.137	1.125	1.163	1.174	1.113	1.181	**1.36**
num_control	1.027	1.0489	1.007	1.034	1.041	1.063	1.076	0.999	1.067	**1.137**
num_plasma	1.34	1.522	1.015	**12.88**	1.155	1.395	1.300	1.000	1.265	3.031
obs_error	1.28	1.411	1.235	**2.280**	1.186	1.411	1.522	1.167	1.260	1.561
obs_info	1.064	1.1325	0.952	**2.033**	1.064	1.1325	1.234	1.000	1.162	1.296
obs_spitzer	**1.285**	1.211	1.021	1.010	1.010	1.016	1.000	0.961	1.086	1.279
obs_temp	1.002	1.029	1.011	1.002	1.002	1.029	1.010	0.978	1.045	**1.123**
Harmonic	1.068	1.068	1.061	**1.328**	1.054	1.061	1.061	1.040	1.065	1.13

On the thirteen datasets, the customized compressors outperformed all other algorithms on six datasets. On the remaining data sets, they are outperformed by only one of the nine other algorithms. FPC substantially outperformed custom on four datasets because it uses much larger internal table sizes. Though bzip2 outperformed custom on two datasets, there is not much difference between the ratios. DFCM outperformed custom on msg_bt by a small margin.

Interestingly, custom achieves good compression ratios on the datasets on which the rest of the algorithms are not performing well. For example, on msg_lu, custom provides a thirty percent higher compression ratio than all of the other algorithms. Similarly, on num_comet, it outperforms the other algorithms by twenty percent and on obs_temp by around twelve percent. Of the studied algorithms, only custom provides significant compression on every tested dataset.

Although my approach outperformed FPC on nine datasets, it does not achieve the highest harmonic mean compression ratio as FPC outperforms all other algorithms by a large margin on four datasets. However, my algorithm has a very good harmonic mean compression ratio compared to the other algorithms.

6.3 Throughput

This section examines the compression and decompression throughputs in megabytes per second (i.e., the dataset size divided by the runtime). Table 6.3.1 shows the compression throughputs of seven algorithms including my approach.

Table 6.3.1 Compression throughput comparison

Trace	Compression throughput						
	BZIP2	GZIP	LZOP	FPC	pFPC	gFPC	Custom
msg_bt	6.451	24.11	**619.512**	104.527	154.501	74.772	58.878
msg_lu	6.224	22.491	**485.827**	110.415	126.781	75.183	29.831
msg_sp	6.531	23.776	**652.482**	105.707	159.573	77.097	77.075
msg_sppm	7.881	104.314	473.31	269.504	**597.753**	107.518	160.338
msg_sweep3d	4.353	18.724	**666.667**	187.207	377.358	62.533	51.086
num_brain	6.519	18.311	**666.502**	50.884	97.972	44.861	89.96
num_comet	3.906	16.474	**620.606**	38.266	73.722	37.034	95.522
num_control	6.428	25.658	**664.192**	64.313	78.442	57.116	62.031
num_plasma	7.986	40.557	**440.789**	281.513	118.794	69.072	63.567
obs_error	7.692	34.537	**218.015**	110.019	194.426	67.463	61.387
obs_info	6.662	28.504	**646.429**	108.383	76.695	63.287	39.434
obs_spitzer	7.661	27.563	**252.674**	75.904	109.438	58.154	92.465
obs_temp	6.422	25.468	**635.000**	55.378	78.557	47.27	61.551
Mean	6.515	31.578	**541.692**	120.155	172.616	64.72	65.435

My approach outperforms bzip2 and gzip on all datasets and performs better than FPC on four datasets. It outperforms pFPC and gFPC on one dataset. The compressions throughputs of my approach are close to FPC, pFPC, and gFPC on all datasets except msg_bt and msg_lu. pFPC is faster as it uses a parallel approach. Note that my objective was to achieve a good compression ratio. By using fewer components, custom can be made faster while sacrificing some compression ratio.

Figure 6.1 provides a graphical comparison of the compression throughputs of FPC, pFPC, gFPC, LZOP, and my approach on the thirteen datasets. LZOP is the fastest approach in almost all cases, but it provides one of the worst compression ratios. The mean throughput of my approach is better than bzip2 and gzip and is almost equal to gFPC. FPC and pFPC outperform my version.

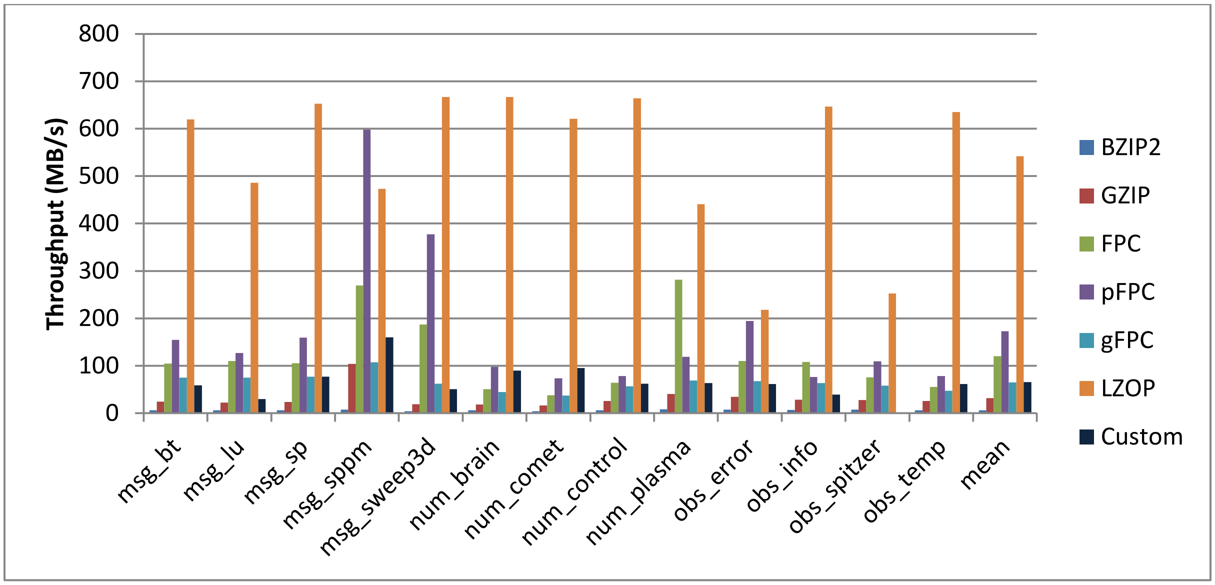

Figure 6.1 Compression throughput

Table 6.3.2 and Figure 6.2 below compare the decompression throughputs of different algorithms. As shown in the table, my approach decompresses the files faster than bzip2 on all datasets. As compression and decompression are largely symmetric in my approach, the decompression time is close to the compression time whereas several of the other algorithms have noticeably higher decompression throughputs than compression throughputs.

The last row represents the arithmetic mean values of all the algorithms on the datasets. LZOP has the highest mean and my approach outperforms bzip2.

Table 6.3.2 Decompression throughput comparison

Trace	Decompression throughput						
	BZIP2	GZIP	LZOP	FPC	pFPC	gFPC	Custom
msg_bt	16.279	114.157	460.980	175.293	**725.714**	644.670	75.192
msg_lu	14.712	105.051	383.230	97.267	**443.885**	642.708	35.727
msg_sp	15.680	104.258	432.344	189.521	**700.506**	641.995	60.100
msg_sppm	52.662	229.793	320.602	383.983	**791.964**	673.671	102.425
msg_sweep3d	10.717	77.569	685.714	265.487	**810.811**	563.380	43.306
num_brain	14.899	108.587	**329.197**	96.505	151.512	182.345	59.420
num_comet	15.817	114.158	**613.174**	91.921	249.756	158.760	100.986
num_control	15.270	117.543	**551.087**	73.195	128.789	185.488	35.923
num_plasma	16.742	131.890	229.452	265.873	**531.746**	179.144	64.547
obs_error	16.818	121.020	**329.444**	123.800	279.717	277.103	68.634
obs_info	14.946	115.287	**624.138**	148.361	489.189	489.189	29.672
obs_spitzer	15.884	98.643	**317.647**	72.860	173.554	147.887	51.724
obs_temp	15.131	114.414	**552.174**	90.499	508.000	501.316	39.037
Mean	18.120	119.413	**448.399**	159.582	460.396	406.743	58.976

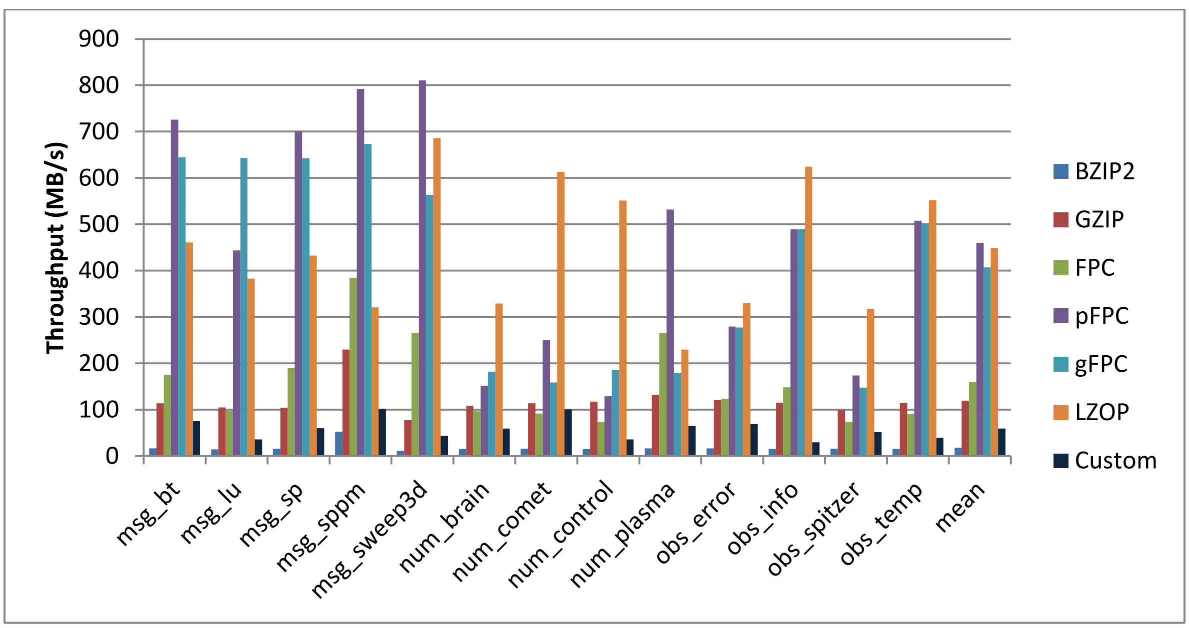

Figure 6.2 Decompression throughput

6.4 Exhaustive Search

This section compares the exhaustive search results of up to three stages with the genetic algorithm results. Table 6.4.1 represents the compression ratios of the exhaustive and genetic approaches for 1, 2, and 3 stages. Exhaustive and genetic yield same results with 1 and 2 stages, i.e., the genetic approach finds the best solution when just a few stages are used. With three stages, exhaustive found a better solution than genetic on two datasets. The genetic algorithm finds this solution when it is allowed to run for more than ten generations.

Table 6.4.1 Comparison of genetic and exhaustive

Trace	Exhaustive			Genetic			Genetic over Exhaustive		
	1	2	3	1	2	3	1	2	3
msg_bt	1.062	1.206	1.229	1.062	1.206	1.229	1.000	1.000	1.000
msg_lu	1.000	1.170	**1.318**	1.000	1.170	1.255	1.000	1.000	0.912
msg_sp	1.000	1.219	1.255	1.000	1.219	1.255	1.000	1.000	1.000
msg_sppm	4.141	5.179	5.544	4.141	5.179	5.544	1.000	1.000	1.000
msg_sweep3d	1.017	1.215	1.292	1.017	1.215	1.292	1.000	1.000	1.000
num_brain	1.000	1.158	1.214	1.000	1.158	1.214	1.000	1.000	1.000
num_comet	1.081	1.254	1.307	1.081	1.254	1.307	1.000	1.000	1.000
num_control	1.013	1.096	1.128	1.013	1.096	1.128	1.000	1.000	1.000
num_plasma	1.063	1.281	2.258	1.063	1.281	2.258	1.000	1.000	1.000
obs_error	1.229	1.289	**1.342**	1.229	1.289	1.338	1.000	1.000	0.997
obs_info	1.006	1.200	1.252	1.006	1.200	1.252	1.000	1.000	1.000
obs_spitzer	1.039	1.216	1.251	1.039	1.216	1.251	1.000	1.000	1.000
obs_temp	1.000	1.095	1.120	1.000	1.095	1.120	1.000	1.000	1.000

6.5 Pure Entropy Based Compression

Table 6.5.1 compares compression ratios of each dataset that could be achieved using pure entropy based compression algorithms before and after using my approach. It is clear from the table that after using my approach, the compression ratio that could be achieved further is very less which represents the effectiveness of the algorithm.

Table 6.5.1 Compression ratio using pure entropy based compression

Dataset	Compression ratio that could be achieved	
	Before	After
msg_bt	2.703	1.024
msg_lu	2.615	1.000
msg_sp	2.556	1.031
msg_sppm	5.690	1.048
msg_sweep3d	2.733	1.003
num_brain	2.670	1.002
num_comet	2.903	1.002
num_control	2.651	1.000
num_plasma	4.688	1.012
obs_error	3.595	1.002
obs_info	3.541	1.004
obs_spitzer	3.686	1.002
obs_temp	2.876	1.000

6.6 Discussion

This subsection discusses some of the above results in more detail.

6.6.1 Best Combinations

The best combinations of ten components for each dataset that the genetic algorithm yielded are presented in Table 6.6.1. The results show that expanders and predictors tend to be used in the first stages for predicting the values. They are followed by shufflers to rearrange the elements to make compression easier. Finally, reducers perform the actual compression operations. In some cases, predictors are directly followed by a reducer and sometimes predictors are used after a first reducer.

Table 6.6.1 Best combinations

Dataset	Best combination
msg_bt	hLNV6s LZB4 | DIM8 LZB3
msg_lu	DIM5 hPLY7s DIM8 LZB2 ZE PLY1s | PLY3x LZB4
msg_sp	FCM7s hLNV6s LZB4 | DIM8 LZB3
msg_sppm	LZB5 LZB4 LNV1x SEL3 MSB SEL7 BIT | NEG LZB2
msg_sweep3d	NEG DFCM6s DIM4 LNV8s DIM8 LZB1 BIT | DIM2 DIM2 LZB2
num_brain	LNV1s LNV2s BIT | LNV4x LZB2
num_comet	LNV1s NEG RLEa DIM7 BIT DIM64 DIM4 | LZB3
num_comet	LNV1s NEG RLEa DIM7 BIT DIM64 DIM4 | LZB3
num_plasma	LNV2s hLNV4s SEL0 | NEG ZE DIM2 RLEb LNV1x MSB LZB5
obs_error	DIM8 LZB2 DIM3 LZB1 BIT DIM64 LNV8x LNV8x | RLEb
obs_info	hLNV2s DIM2 BIT DIM64 INV | RLEb DIM2 LNV1x LZB3
obs_spitzer	ZE MSB LNV1x BIT DIM64 DIM4 | LZB2
obs_temp	LNV8s DIM4 BIT INV | LNV4x LZB2 LZB5

6.6.2 Repeated Combinations

Below I list some combinations of components that occur often.

BIT → LZB and RLE

As BIT groups the n^{th} bit values of each element in a chunk together, it is likely that the output sequence has repeated values. These values can then be compressed easily using reducers, which is why BIT is mostly followed by a reducer.

DIM → LZB

This sequence occurs after a few predictors and shufflers. The reason for this is that DIM groups values from the same dimension together and, due to the predictors and shufflers employed before this component, there again is a higher chance of repetitions. These repetitions can then be compressed using the best reducer, which is LZB.

LZB → LZB

There are different versions of the LZB component that target different patterns. Hence, it is sometimes necessary to use more than one such LZB component to capture the redundancy in the data.

6.7 Customization Benefits

Customization can provide a tailored algorithm for a specific file if desired. Increasing the number of generations may yield better compression ratios. Also, the customization approach described in this thesis can be applied to other types of data. New components can be added easily to potentially improve the compression ratio and components can be removed to speed up the search.

CHAPTER 7

SUMMARY

This thesis describes an approach to automatically synthesize a tailored compression and decompression algorithm for a given input file. The algorithms are built by chaining algorithmic components that were extracted from pre-existing lossless compression algorithms. Each algorithmic component has an inverse component that performs the opposite action, making it possible to automatically generate a de-compressor for each synthesized compressor. Exhaustive search and a genetic algorithm are used to find the best possible algorithm in the search domain. The presented approach makes it easy to add additional components and can be applied to different data domains.

When tested on thirteen difficult-to-compress real-world double-precision floating-point datasets, the synthesizes algorithms I found yield the highest harmonic mean compression ratio among X tested algorithm and is only outperformed by FPC. My algorithms deliver a throughput of 65 MB/s for compression and 59 MB/s for decompression.

LITERATURE CITED

2006. http://www.llnl.gov/asci/purple/benchmarks/limited/code_list.html.

Bailey David, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo and Maurice Yarrow. *The NAS Parallel Benchmarks 2.0.* Technical, Moffett Field: NASA Ames Research Center, 1995.

Burtscher Martin and Sam Nana B. "TCgen: A tool to automatically generate lossless trace compressors." *ACM SIGARCH Computer Architecture News*, 2006: 1-6.

bzip2. 2006. http://www.bzip.org/.

Vadim Engelson, Fritzson Dag and Fritzson Peter. "Lossless Compression of High-volume Numerical Data from Simulations." *Data Compression Conference.* Snowbird, UT: IEEE, 2000. 574-586.

Fang Webin, Bingsheng He and Qiong Luo. "Database Compression on Graphic Processors." *Proceedings of the VLDB Endowment*, 2010: 670-680.

Frigo Matteo and Steven G. Johnson. *The Fastest Fourier Transform in the West.* Technical, Massachusetts: Massachusetts, 1997.

Garber Brian A, Dan Hoeflinger, Xiaoming Li and Maria.Jesus. Garzaran. "Automatic generation of parallel sorting algorithms." *Parallel and Distributed Processing.* Miami, FL: IEEE, 2008. 1-5.

Genetic Algorithm. n.d. http://en.wikipedia.org/wiki/Genetic_algorithm.

William H. Hsu and Amy E Zwarico. "Automatic synthesis of compression thechnique for heterogeneous files." *Software - Practice and Experience* , 1995: 1097-1116.

Huffman, David A. "A method for the construction of minimum redundancy codes." *Proceedings of the Institute of Radio Engineers*, 1952: 1098-1101.

Kattan Ahmed and Poli Riccardo. "Evolutionary synthesis of lossless compression algorithms with GP-zip3." *IEEE*, 2010: 1-8.

Langdon Glen G. "An Introduction to Airthmetic Coding." *IBM Journal of Research and Development*, 1984: 135-149.

Lempel Abraham and Ziv Jacob. "A Universal Algorithm for Sequential Data Compression." *IEEE Transactions of Infromation Theory,* 1977: 337-342.

Lindstorm Peter and Isenburg Martin. "Fast and Efficient Compression of Floating-Point data." *IEEE Transactions on Visualization and Computer Graphics*, 2006: 1245-1250.

lzop. n.d. http://www.lzop.org.

Burtscher Martin and Ratanaworabhan Paruj. "FPC: A High-Speed Compressor for Double-Precision Floating-Point Data." *IEEE Transactions on Computers*, 2008: 18-31.

Burtscher Martin and Ratanaworabhan Paruj. "gFPC: A Self-Tuning Compression Algorithm." *Data Compression Conference,* 2010: 396-405.

Burtscher Martin and Ratanaworbhan Paruj. "pFPC: A Parallel Compressor for Floating-Point Data." *Data Compression Conference.* Snowbird: IEEE, 2009. 43-52.

Suman K. Mitra, C. A. Murthy, and Kundu Malay K. "Technique for Fractal Image Compression using Genetic Algorithm." *IEEE Transactions on Image Processing*, 1998: 586-593.

Shanmugasundaram Senthil and Lourdusamy Robert. "A Comparative Study of Text Compression Algorithms." *International Journal of Wisdom Based Computing*, 2011: 68-76.

The gzip home page. 2006. http://www.gzip.org/.

Ziv Jacob, and Lempel Abraham. "Compression of Individual Sequences via Variable-Rate Coding." *IEEE Transactions of Information Theory*, 1978: 530-536.

Sayood Khalid. *Introduction to Data Compression.* San Francisco, CA: Morgan Kauffmann, 2012.