Increasing Telemetry Throughput Using Customized and Adaptive Data Compression

Jared Coplin[1] coplin@txstate.edu
Annie Yang[1] ayang@txstate.edu
Andrew R. Poppe[2] poppe@ssl.berkeley.edu
Martin Burtscher[1] burtscher@txstate.edu
[1]Department of Computer Science, Texas State University
[2]Space Sciences Laboratory, University of California, Berkeley

ABSTRACT

Due to the increasing generation of massive amounts of data by space-based instruments, it has become a challenge to transmit even a fraction of a typical spacecraft data volume back to Earth in a feasible amount of time. Thus, improvements in the ability to losslessly compress data on-board before transmission represent an important method of increasing overall data return rates. We describe a custom methodology for compressing spacecraft data on-board that provides significant improvements in both compression ratio and speed. We have used data returned by the five-probe THEMIS/ARTEMIS constellation to quantify the compression ratio and compression speed improvements over a variety of data types (e.g., time-series and particle data). Our approach results in a 38% improvement in compression ratio and up to a three-fold improvement in compression throughput and energy efficiency. We argue that such methods should be adopted by future space missions to maximize the data return to Earth, thus enabling greater insight and scientific discovery.

1. INTRODUCTION

Scientific endeavors in space are important in terms of exploring and answering questions such as how the universe came to be and where life began. They also play key roles in protecting people by tracking storms, tsunamis, volcanic eruptions, etc. All of them depend on high-quality data, and high-quality data require advanced instruments to collect, process, and return those data.

Higher resolution instruments are constantly being developed for use in space. While such instruments may yield higher quality data than previous generations, the amount of data to be returned to Earth for processing increases significantly as the resolution improves. The general solution to this problem is to compress the data, but the challenges to do so well and quickly in the power, memory, and compute-capability constrained environment of spacecraft are significant. Any data that cannot be transmitted are typically lost. This is already a major problem and will likely get worse in the future.

To improve this situation, we propose to automatically synthesize customized compression algorithms that are tailored to a specific type of data. The resulting lossless algorithms not only yield high compression ratios but also high throughputs. This approach increases the data return rate while simultaneously preserving the resolution of data when sending them back to Earth.

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

Many existing and proposed spacecraft contain multiple instruments possibly operating in several modes, posing significant challenges when it comes to finding a compression algorithm that works well on all the various data packets generated by these instruments. Furthermore, transmitting data is relatively slow and consumes a significant amount of energy, especially for interplanetary missions, so we desire compression algorithms that not only minimize the number of bits that need to be transmitted but also process data rapidly. The compression needs to be fast to avoid the time and energy cost of sending uncompressed data, or losing data due to being unable to send them in the timeframe allocated for transmission. This compression should be done losslessly to preserve the full integrity of the data. Since good data models are domain and instrument specific, custom designing an efficient algorithm for each packet type is considered to be overly complex, expensive, and error prone.

This has led to spacecraft being launched with just a few compression algorithms programmed into the hardware or software that need to handle data from many different instruments and modes. However, these algorithms are still not tailored to each instrument and mode. This results in sub-optimal telemetry throughputs, which in turn increases the energy expenditure of transmitting data or results in data loss. Moreover, data tend to change over time, and sometimes spacecraft are repurposed at the end

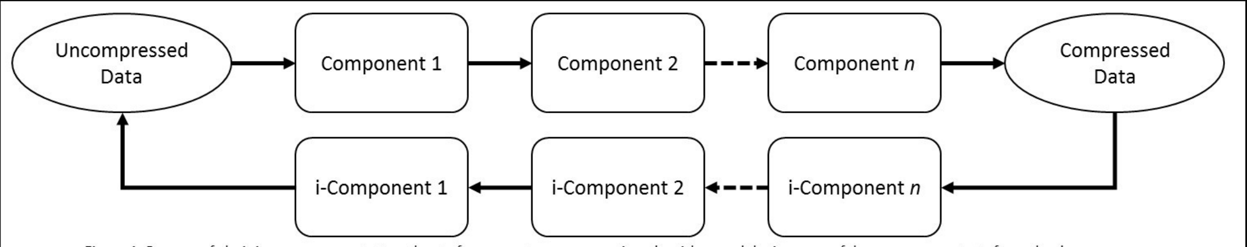

Figure 1: Process of chaining *n* components together to form a custom compression algorithm, and the inverses of those components to form the decompressor

of their main mission. Preprogrammed compression algorithms are not able to adapt to such changes. What is more, current algorithms are not designed for maximum processing speed but typically focus more on achieving higher compression ratios. This leads to situations where not all data can be compressed in the time available before they need to be transmitted.

Ultimately, the biggest problem with suboptimal telemetry throughput is that it lowers the cost effectiveness of space probes. If a craft cannot return all the data it is designed to collect, some of the time and money put into building, testing, and launching the craft is wasted.

With these issues in mind, we propose automatically synthesizing custom compression algorithms from a library of interoperable *algorithmic components*. These components are the result of a thorough analysis of previously proposed compression algorithms. In particular, we broke many prior algorithms down into their constituent parts, rejected all parts that could not be implemented in a simple and fast way, and generalized the remaining components as much as possible. We then implemented the 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 makes it possible to *chain* the components, allowing for the generation of a large number of compression algorithm candidates from a given set of components. We implemented this approach in a synthesis framework called SDcrush, which is capable of automatically determining the best chain of components (aka compression algorithm) for a given set of training data. For each algorithmic component, SDcrush includes an inverse that performs the opposite transformation, enabling it to also automatically generate the corresponding decompression algorithm. Figure 1 illustrates this approach. Section 2 describes all available algorithmic components in detail.

For synthesizing a custom algorithm, one possible approach is to use SDcrush during spacecraft or instrument design using training data. How varied the types of data are determines how many separate compression algorithms are needed. For instance, if there are *m* different types of data packets, SDcrush can be used to generate *m* algorithms to maximally compress all the data.

Alternatively, the spacecraft could be launched with a software library of compression components akin to SDcrush's database. This way, a custom algorithm can be synthesized on the ground based on recent data. For example, on Wednesday, the spacecraft gathers data and compresses them using an algorithm that had been uploaded on Tuesday and was synthesized based on data received on Monday. On Thursday, the craft compresses the data using an algorithm that had been uploaded Wednesday based on the Tuesday data and so on. Figure 2 illustrates this approach. This allows the craft to adapt to changes in the data.

This paper makes the following main contributions:

1) It proposes the chaining of compression algorithm components to synthesize novel lossless compression algorithms that can significantly improve data reduction, compression throughput, and energy efficiency.

2) It investigates three progressively more adaptive approaches for using data compression to improve telemetry throughput on spacecraft, each with its own advantages and drawbacks.

3) It shows that when the proposed compression techniques are used aboard spacecraft, a 38% increase in compression ratio and a three-fold increase in compression throughput and energy efficiency can be achieved.

The remainder of this paper is organized as follows: Section 2 details the algorithmic components SDcrush uses, Section 3 explains our approach, Section 4 discusses related work, Section 5 describes our evaluation methodology, Section 6 presents and analyzes our experimental results, and Section 7 summarizes our findings and draws conclusions.

2. ALGORITHMIC COMPONENTS

This section describes the 36 algorithmic components currently in SDcrush for synthesizing compression algorithms. Each component takes a sequence of values as input, transforms it, and outputs the transformed sequence. To organize the description of the components, we grouped them into categories.

2.1 Shufflers

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

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

The **ROT*n*** component takes a parameter *n* that specifies by how many units to rotate the bits of each word in the input sequence. There are seven versions of this component. This rotation affects the behavior of some of the other components.

The **DIM*n*** component takes a parameter *n* that specifies the dimensionality of the input sequence and groups the values accordingly. For example, a dimension of three changes the linear sequence x_1, y_1, z_1, x_2, y_2, z_2, x_3, y_3, z_3 into x_1, x_2, x_3, y_1, y_2, y_3, z_1, z_2, z_3. This may be beneficial as values belonging to the same dimension often correlate more with each other than with values from other dimensions. We use $n = 2$, 3, 5, 7, 8, 12, and 16. Since a dimensionality of $k{\cdot}m$ can be represented by combining a DIM*k* with a DIM*m* component, we primarily use small prime numbers for the parameter *n*. To capture important non-prime values of *n* in a single component (for performance reasons), we also include the following: $n = 12$ is useful because twelve is the least common multiple of 2, 3, 4, and 6 and therefore works well on 2D, 3D, 4D, 6D, and 12D data. $n = 8$ is useful in combination with the BIT component because there are eight bits per byte.

The **SWP** component reverses the byte order of every other value, thus bringing the lower bytes of consecutive values closer together as well as the higher bytes of the next pair of values and so on, which might improve compressibility as similar data tend to be compressed better when they are in close proximity.

2.2 Predictors

Predictors guess the current value based on previous values in the input sequence, subtract the predicted from the current value, and emit the result of the subtraction, i.e., the residual sequence. If the predictions are close to the actual values, the residuals will cluster around zero, making them easier to compress than the original sequence. Predictors do not change the length of the sequence.

The **LVs** and **LVx** components use the previous value in the sequence as a prediction of the current value. This is commonly referred to as delta modulation. The subtraction to compute the residual can be performed at word granularity (using conventional subtraction, denoted by a trailing '**s**') or at bit granularity (using XOR, denoted by a trailing '**x**').

2.3 Reducers

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

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

The **RLE** component performs run-length encoding. In particular, it counts how many times the current value appears in a row. Then it counts how many non-repeating values follow. Both counts are recorded in a single word, i.e., each count gets half of the bits. This "count" is emitted first, followed by the current value and finally the non-repeating values. This procedure repeats until the end of the input is reached.

The **LZ*ln*** component implements a variant of the LZ77 algorithm [1]. It uses a hash table to identify the *l* most recent prior occurrences of the current value. Then it checks whether the *n* values immediately before those locations match the *n* values just before the current location. If they do not, the current value is emitted and the component advances to the next value. If the *n* values match, the component counts how many values following the current value match the values following that location. The length of the matching substring is emitted and the component advances by that many values. Smaller values of *n* yield more matches, which have the potential to improve compression but also result in a higher chance of zero-length substrings, which hurt compression. Larger values of *l* improve the chance of finding a match but results in slower processing. We consider $n = 1$, 2, 3, 4, 5, 6 and 7 combined with $l =$ a, b, and c, where a = 1, b = 2, and c = 4, which gives us 21 different LZ*ln* components. They each contain a hash table with 8192 two-byte elements (i.e., 16 kB).

2.4 The NUL Component

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

2.5 The Cut

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

2.6 Component Discussion

The components described above can all be implemented to run in O(n) time, where n is the sequence length. We excluded more sophisticated components such as block-sorting components to make the synthesized algorithms as fast as possible. Nevertheless, as the results in this paper demonstrate, the included components suffice to create fast algorithms that compress competitively.

Due to the presence of the Cut, SDcrush needs two versions of each component: one for words and one for bytes. We implemented all components using C++ templates to facilitate the generation of these versions.

Each component requires an inverse component that performs the reverse transformation. By chaining the inverse components in the opposite direction, SDcrush can automatically synthesize the matching decompression algorithm for any given chain of components, i.e., for any compression algorithm it can generate.

Using algorithmic components greatly simplifies testing and verification. Since the components are independently implemented, it suffices to validate them in isolation, rather than all combinations of components, to ensure that any set of components will function properly.

The 21 LZ*ln* components utilize hash tables. For performance reasons, their hash functions only use some of the bits from the input values. This is why altering the location of bits and bytes by other components affects the effectiveness of these components. Note, however, that SDcrush is able to optimize which bits to use by the hash function, for example, through the inclusion of an appropriate ROT component.

Not counting the Cut, SDcrush has 36 components at its disposal, 23 of which are able to reduce the length of the data. The purpose of the remaining 13 components is to transform the values in such a way that the reducers become maximally efficient. Thus, longer chains of components have the potential to compress better but make the search for a good algorithm take longer and compression and decompression slower. For an algorithm with k stages, that is, a chain with k components, the search space encompasses $(k+1) \cdot 36^{k-1} \cdot 23$ possible algorithms because there are $k+1$ locations for the Cut, $k-1$ stages that can each hold any one of the 36 components, and a final stage that can hold any one of the 23 reducers. For example, this amounts to over 231 million possible five-stage compression and decompression algorithms.

3. APPROACHES

We use SDcrush to perform an exhaustive search to determine the most effective spacecraft-friendly compression algorithms that can be created from the available components for the following three scenarios, which even work on a spacecraft in the outer solar system, where it may take several hours for a signal to travel round trip.

3.1 Unified

Unified uses a single algorithm for the entire spacecraft. The advantage of this approach is that it is the simplest. A spacecraft can be launched with just one algorithm, possibly implemented in hardware, which it will use all the time no matter what type of data are being compressed. Of course, the downside is that it will result in lower compression ratios compared to more specialized approaches.

3.2 Packet

Packet is similar to *Unified* in that it employs fixed algorithms. However, *Packet* uses a different algorithm for each packet type. The advantage of having a custom algorithm per packet type is better compression. The downside is that it requires more complex hardware or software to implement all the needed algorithms.

3.3 Adaptive

Both of the previous approaches assume that the training data used to determine an effective fixed algorithm on the ground before the spacecraft is launched is highly representative of the data that the craft is expected to collect in space. If this is not the case, or if the spacecraft is later repurposed, the fixed algorithms may not be able to compress the data effectively, reducing telemetry throughput without the possibility to change the algorithm.

Adaptive solves this problem. It is a previous n day approach, where the data used to determine the compression algorithm to be used was generated n days ago. In this paper, we show results for $n = 2$. In other words, today we are compressing data using a custom algorithm that is synthesized based on actual spacecraft data from two days ago. Figure 2 illustrates this. *Adaptive* requires the capability of uploading a new compression algorithm for each packet type on a regular basis. Note, however, that there is no need to upload the actual algorithm but only a short sequence (chain) of component identifiers.

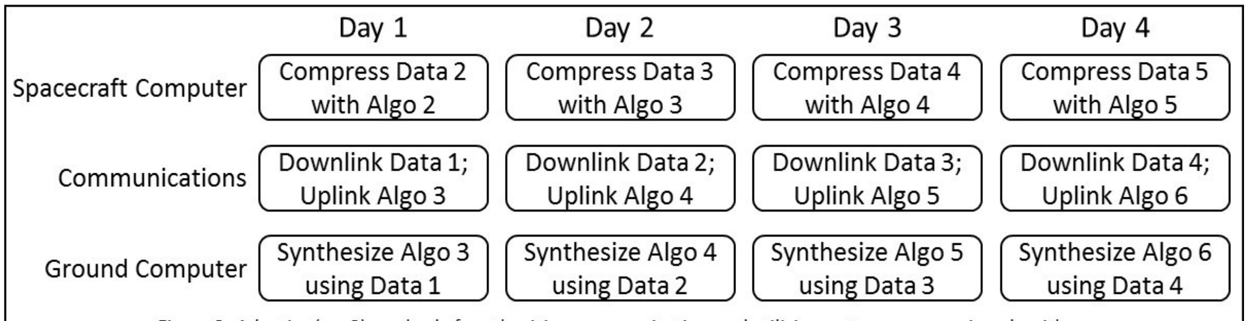
Day 1
Day 2
Day 3
Day 4
Spacecraft Computer
Communications
Ground Computer
Compress Data 2 with Algo 2
Compress Data 3 with Algo 3
Compress Data 4 with Algo 4
Compress Data 5 with Algo 5
Downlink Data 1; Uplink Algo 3
Downlink Data 2; Uplink Algo 4
Downlink Data 3; Uplink Algo 5
Downlink Data 4; Uplink Algo 6
Synthesize Algo 3 using Data 1
Synthesize Algo 4 using Data 2
Synthesize Algo 5 using Data 3
Synthesize Algo 6 using Data 4

Figure 2: *Adaptive* ($n = 2$) method of synthesizing, communicating, and utilizing custom compression algorithms

This amounts to just a few bytes of data per algorithm, which is in line with other configuration information that is uplinked such as changes to the mode in which the onboard instruments operate. Of course, nothing needs to be sent if there is no change in the algorithm to be used.

4. RELATED WORK

To our knowledge, this is the first study to use algorithmic components chained together to synthesize custom compression algorithms that are spacecraft friendly.

The recognition of the need for adaptive data compression dates back several decades to research conducted by Rice and Plaunt in preparation for a grand-tour mission of the outer planets [2]. Their work centers around compression of images, but the overarching concept can be applied to other forms of data compression. Rice et al. later proposed a very simple algorithm that chooses from many "Huffman equivalent" codes. It incorporates a built-in preprocessor that acts like the predictor component outlined in Section 2.4 and can include external predictors as well [3]. This work was later recommended to the Consultative Committee for Space Data Systems (CCSDS) as the basis for their standard with added low entropy options [4][5][6]. As the compression efficiency of this approach decreases with increasing numbers of outliers in the data, Portell et al. designed the Fully Adaptive Prediction Error Coder, which performs well even in the presence of outliers [7]. Algorithms designed for specific instruments such as the Synthetic Aperture Radar (SAR) [8] and plasma detection instruments [9] have also been proposed.

Using SDcrush to generate compression algorithms differs from these approaches in that it can synthesize a custom algorithm for any given data set. So rather than attempting to design an algorithm that is based on a model of the data expected to be collected, we find an algorithm based on actual collected data. This method gives us the freedom to entirely change the basis of the compression algorithm if the data require it. We have successfully used a very similar approach to generate a massively parallel floating-point compressor for GPUs [10] as well as tailored floating-point compressors for CPUs that can be synthesized in real time [11].

5. EVALUTION METHODOLOGY

5.1 Data

The data used in our evaluation comes from the THEMIS B spacecraft that is part of the five-probe THEMIS/ARTEMIS constellation currently in orbit around the moon and Earth [12][13]. Each THEMIS probe has six scientific instruments that can operate in multiple modes resulting in 26 distinct types of data packets, which are identified by a set of three hexadecimal digits [14]. The craft also produces packets that contain spacecraft status information. These packets amount to relatively little data and are therefore less interesting from a compression standpoint. We studied all 26 data packet types but only show results for three of them (443, 444, and 44d) as they seem to be quite representative of the behavior of the other packet types.

Our *Unified* and *Packet* approaches use algorithms generated from data collected by THEMIS B in 2013. However, the results we show are from data collected by the same probe in 2014 to avoid gathering results from the same data we used to generate the algorithms.

5.2 Compression Algorithms

5.2.1 Baseline

Each of the packet types we study has their own on-board data compression algorithm. 443 uses 3-channel Delta modulation, 444 uses standard Delta modulation, and 44d uses Huffman compression, respectively. We re-created these compression algorithms to obtain an accurate baseline of comparison to our custom algorithms. We also compare our approaches to the commonly used general-purpose compressors gzip and bzip2 [15][16].

5.2.2 Gzip

Gzip is a lossless data compression tool that uses Lempel–Ziv coding (LZ77) to reduce the size of files [1]. LZ77 algorithms compress by replacing repeated instances of data with references to a single copy of that data that was seen earlier in the data stream.

For gzip, we show results using the --fast and --best flags to show its full range of possible compression ratios, compression throughputs, and energy consumption.

5.2.3 *Bzip2*

Bzip2 compresses data using the Burrows-Wheeler block sorting algorithm as well as Huffman coding [17]. The Burrows-Wheeler transform (BWT) rearranges a character string into runs of similar characters [18]. This makes the data relatively simple to compress using other methods such as run-length encoding. Generally speaking, bzip2 compresses better than gzip in terms of compression ratio, but does so at the cost of decreased throughput.

For bzip2, we only show results using the --fast flag. While bzip2 --best yields very good compression ratios, the throughput and energy consumption are several hundred times worse than the baseline. Moreover, the memory consumption is excessive. For these reasons, we exclude the bzip2 --best results from this study.

5.2.4 *Custom Algorithms*

For each of the studied packet types, we generate algorithms that consist of five components. We found that using more than five components only marginally increases the compression ratio while, at the same time, greatly decreasing the throughput. The NUL component ensures that shorter compression algorithms are also included.

The algorithms are represented by a series of component identifiers. For example, "DIM3 | BIT RLE" is a three-component algorithm composed of DIM with a dimensionality of 3, which is followed by the Cut, the BIT component, and finally the RLE reducer component.

The *Unified* approach concatenates all data from the probe for an entire year, runs them through SDcrush to find the best compression algorithm for those data, and then runs the identified algorithm over the daily data from the following year to gauge the range of compression ratios and throughputs over time if the spacecraft only had one algorithm with which to compress all data.

With *Packet*, we again concatenate all the data for an entire year, but separately for each packet type. We then use the resulting algorithms to compress each individual data packet from the following year to obtain the range of compression ratios and throughputs over time for a particular packet type.

Adaptive uses an on-board software library of algorithmic components as well as some trivial software to chain them together into a compression algorithm. The spacecraft is sent configuration information telling it which components to chain into an algorithm for specific packet types. The configuration sent to the spacecraft is generated automatically based on recently received data that were processed on the ground using SDcrush.

5.2.5 *Metrics*

The compression ratios are the uncompressed data size divided by the compressed data size. However, we do not report the absolute compression ratio but rather relative results, i.e., how much more or less the data are reduced in size compared to the baseline. Results above 1.0 indicate that the corresponding algorithm compresses better than the baseline.

The throughput is the uncompressed packet size divided by the compression runtime. Again, we report results relative to the baseline where values above 1.0 indicate that the corresponding algorithm compresses faster than the baseline.

To measure the amount of energy it takes to compress any given data packet, we use a custom power tool based on the PAPI framework [19]. This tool accesses Model Specific Registers in the CPU to obtain the amount of energy consumed. We normalize these raw energy measurements with respect to the original size of the input data to compute the number of bytes compressed per joule of energy consumed (bytes/J), i.e., the energy efficiency of the approach. As before, the reported values above 1.0 denote that the corresponding algorithm is more energy efficient than the baseline.

5.3 System Information

We ran all our experiments on a single core of a system that has dual 10-core Xeon E5-2687W v3 CPUs running at 3.1 GHz. The host memory size is 128 GB and the operating system is CentOS 6.7. We used the gcc compiler version 4.4.7 with the "-O3 -xhost" flags.

The THEMIS probes use an Intel 8085 CPU. While we do not have access to an Intel 8085, we only run a single thread to better model the behavior of the Intel 8085. Moreover, while gzip and especially bzip2 use far more memory than is available on the 8085, we limit the memory footprint of our customized algorithms to 32 kB, i.e., half of the maximum memory capacity of the 8085. Furthermore, we only synthesize algorithms that can be implemented on an 8-bit processor.

6. EXPERIMENTAL RESULTS

Figures 3, 4, and 5 display the compression ratio in terms of data reduction, compression throughput, and energy efficiency, respectively, for the three studied packets. All results are normalized. The *x*-axis represents an entire year's worth of data, while the *y*-axis shows the results relative to the baseline (the red dotted line at 1.0). Our approaches are denoted by solid lines whereas the baseline, gzip, and bzip2 use dashed lines. Also, the average of all the results over the entire year can be found in the legend of each figure.

The fixed algorithms are as follows:
Unified: "DIM3 LVs | DIM2 RLE LZa3"
Packet 443: "| DIM12 LZa5 BIT DIM8 LZa3"
Packet 444: "DIM3 BIT RLE RLE | LZc5"
Packet 44d: "RLE | ROT3 BIT ZE LZc4"

Notice that the fixed approaches only use a few of the available components. The remaining components are used, however, in the *Adaptive* approach, which shows

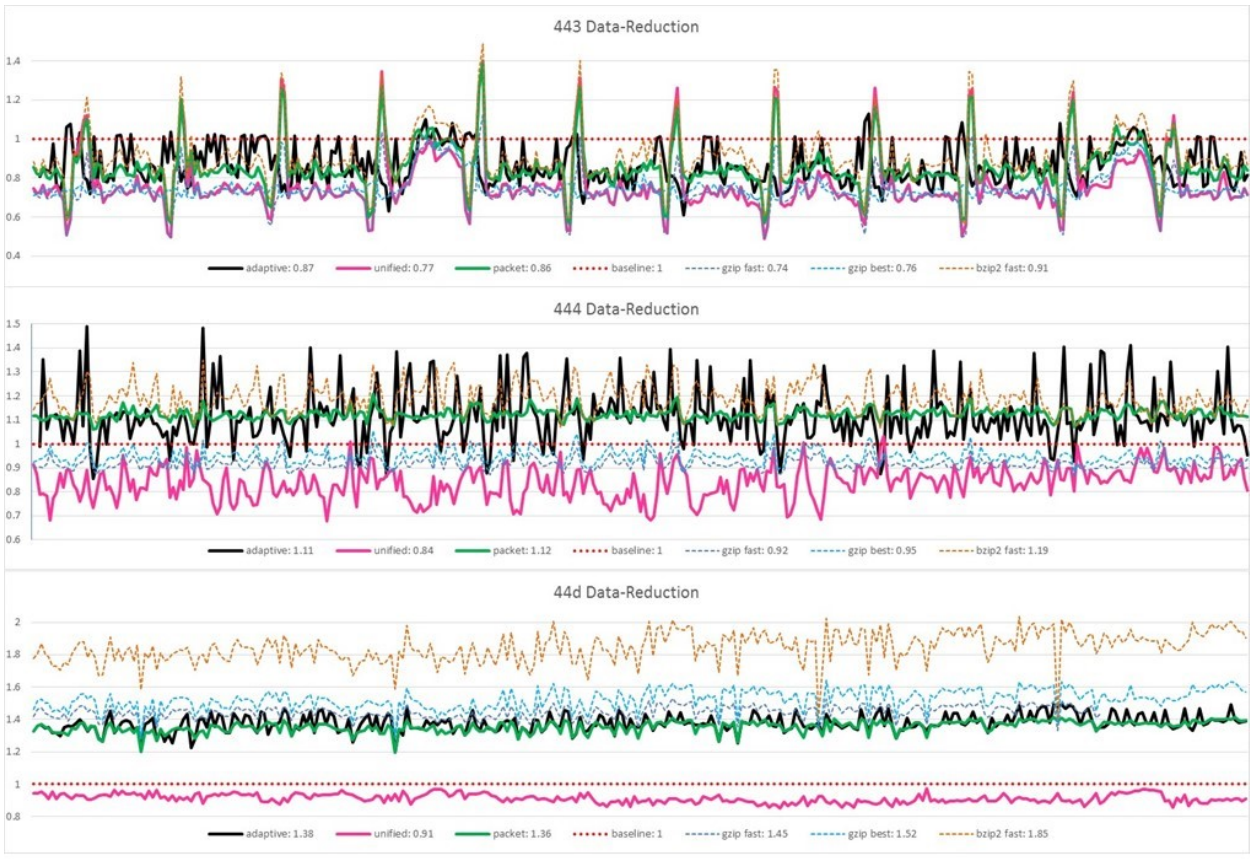

Figure 3: Daily data-reduction over a year for each studied packet type

that there exist circumstances where the more specialized components are beneficial. In fact, about 65% of the *Adaptive* algorithms for packet type 443, 71% for type 444, and 84% for type 44d are completely unique and not used during the rest of the year. However, some of the algorithms do get re-used. The most commonly reused algorithms for each packet type are as follows, though these are used less than 5% of the time:

Adaptive 443: "DIM3 LVs BIT RLE | LZa4"

Adaptive 444: "DIM3 LVs BIT RLE | LZa4"

Adaptive 44d: "NUL NUL NUL | BIT ZE"

Interestingly, 443 and 444 both have the same most reused algorithm while the mostly commonly used *Adaptive* 44d algorithm uses the NUL component three times, meaning it is actually only a two component algorithm that compresses as well or better than all 3, 4, and 5 component algorithms for that particular day's data. Also, about 45% of all the 44d *Adaptive* algorithms use the NUL component at least once.

6.1 Normalized Compression Ratio

Figure 3 shows the amount of data reduction (higher is better) relative to the baseline for each of the three studied packet types. We can see that, while *Unified* does slightly better than baseline on 44d, it does significantly worse than baseline on 443 and 444. This is to be expected as *Unified* is not specialized to the data in different packets but rather to all the data the craft produces.

Looking at the *Packet* results, we find that packet-specific compression improves the 44d compression ratios by an average of about 36% and 444 by roughly 12% over the baseline. However, the *Unified* 443 compression ratios are over 12% worse than the baseline.

We further see that there is less variation in the results with *Packet*. This makes sense as having a specialized algorithm per packet type generally creates fewer outliers than having just one algorithm for the entire craft. The consistency provided by *Packet* could be important by providing a smaller margin of error when estimating the expected compression ratio for a certain packet type while the craft is still in the design and planning phase.

Adaptive performs on par with *Packet* on average. However, there are many individual days where *Adaptive* is much better than *Packet*. Unfortunately, these days are usually immediately followed by a day where the compressed data size is much worse than that of *Packet*. This shows that when *Adaptive* identifies a specialized, atypical algorithm that performs very well, a few days later the compression ratio is negatively affected when the data return to a more typical behavior. We discuss a possible solution to this problem in the Future Work section.

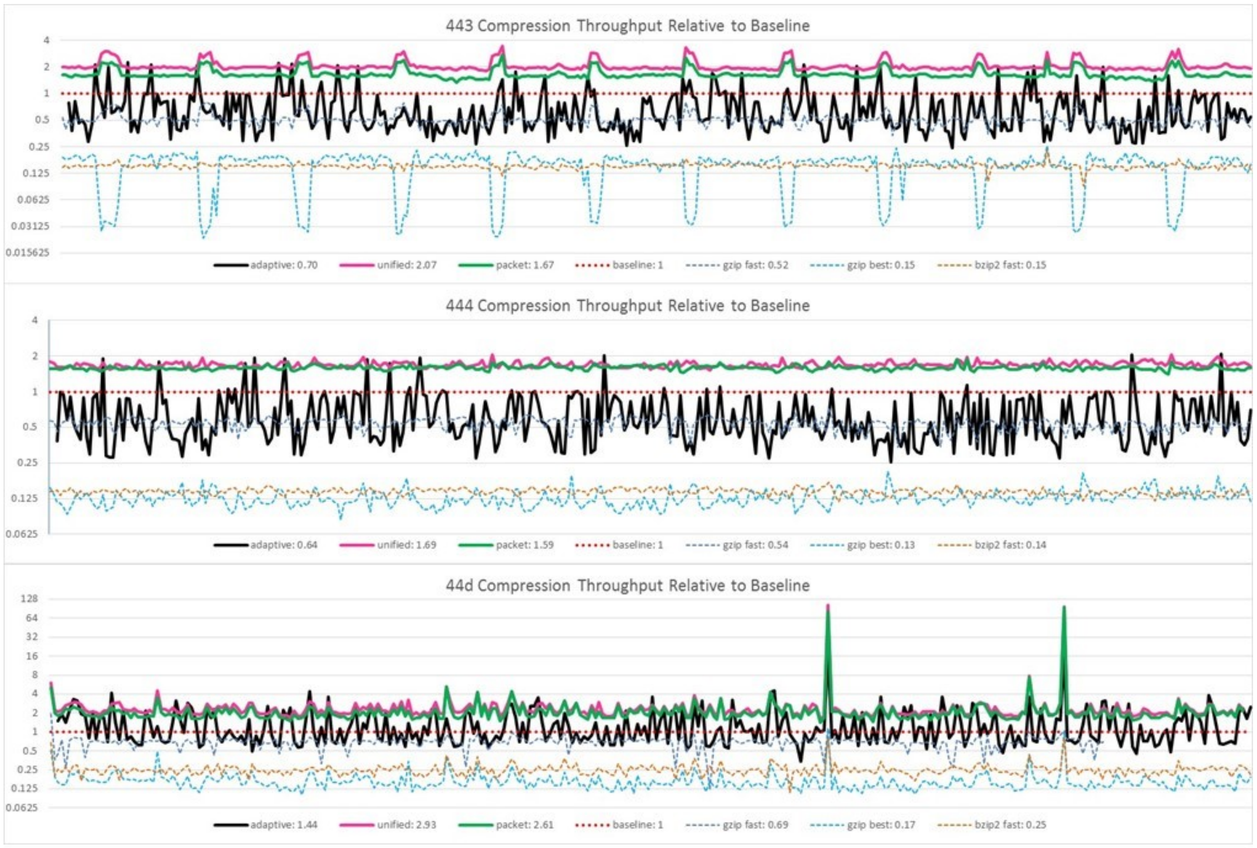

Figure 4: Daily compression throughput relative to the baseline over a year for each studied packet type

Gzip performs slightly better than our custom algorithms for 44d, but worse than both *Packet* and *Adaptive* for 444 and 443. This is despite gzip using more memory and a more complex algorithm. Furthermore, gzip --best is only slightly better than gzip --fast. bzip2, however, compresses better than any of our custom algorithms.

Judging strictly from the size of the compressed packets, bzip2 and gzip are superior to the other approaches. However, we cannot necessarily say they are the best algorithms for compression on a spacecraft because of their relatively complex and memory hungry nature when compared to the baseline or our custom algorithms.

Out of our algorithms, *Packet* and *Adaptive* perform the best for 44d and 444, but *Adaptive* has the advantage of being able to adapt to changing space conditions. For 443, the current baseline compresses better overall than any of the other approaches.

6.2 Normalized Compression Throughput

Figure 4 shows normalized compression throughput results relative to the baseline. This is measured in terms of the number of megabytes compressed per second, so again larger numbers are better. Note that the *y*-axis is logarithmic. We do not show raw runtime figures because those are different on different systems. However, we expect the normalized throughputs to at least be similar across platforms.

Both *Packet* and *Unified* yield a high throughput, with *Unified* outperforming the baseline by roughly a factor of two on all studied packet types. *Packet* and *Unified* are also quite consistent, especially compared to *Adaptive*. Of course, a hardware implementation is likely to result in even higher throughputs.

Surprisingly, *Adaptive* is overall about as fast as the baseline for 44d, but almost 1.5 times worse than the baseline and about four times worse than *Unified* or *Packet* for 444 and 443. We believe this unexpected result is because *Unified* and *Packet* are forced to use simple and general components that work well across many different types of data, while *Adaptive* has the freedom to use more specialized components that reduce throughput in favor of a better compression ratio.

Gzip and bzip2 perform anywhere from 55% to over nine times worse than the baseline in all cases. Again, this is expected as these algorithms are designed to maximize compression at the cost of speed. The outliers on the right side of Figure 4 for 44d show that, for just two days out of the year, the baseline took much longer to compress the data. These outliers do not appear in Figure 3 because the longer runtime does not affect the size of the compressed packet. Clearly, *Unified* is the best approach in terms of consistent throughput, followed closely by

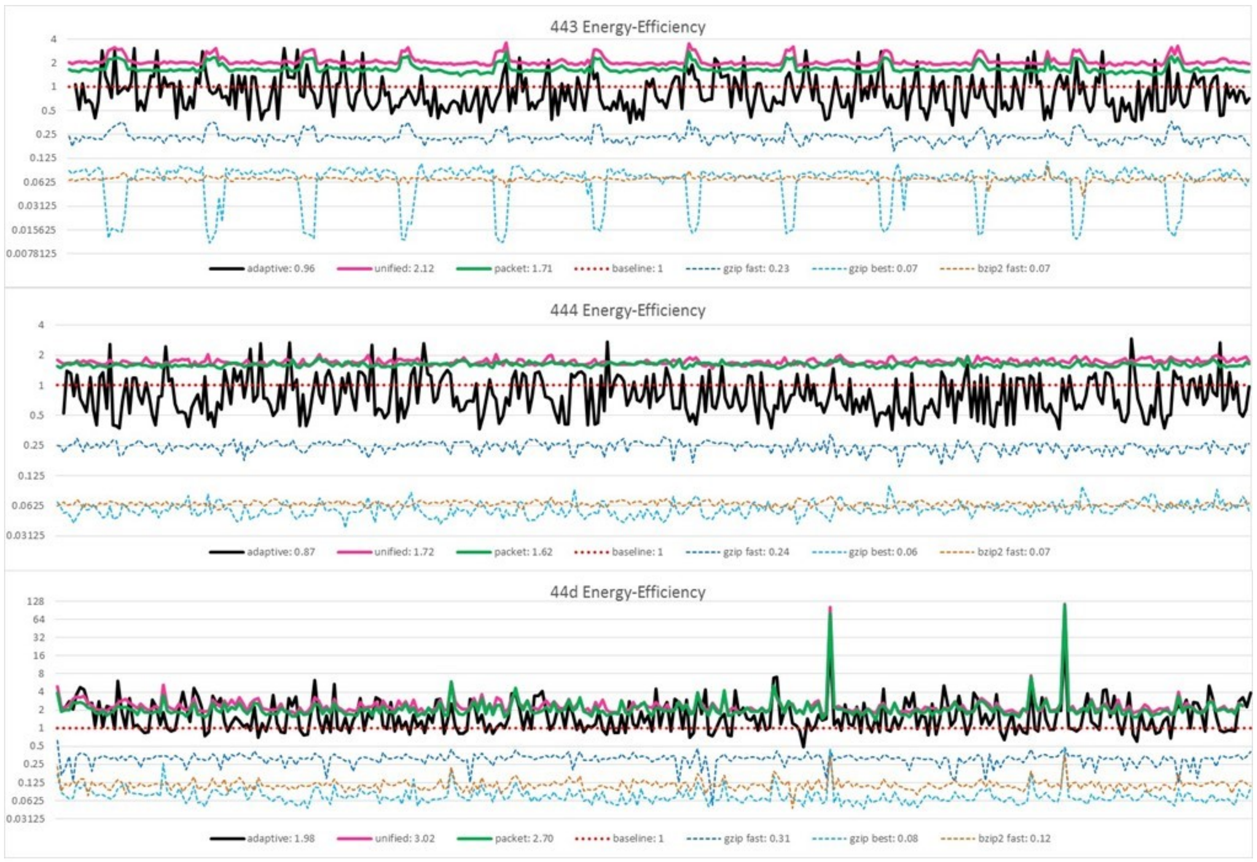

Figure 5: Daily energy-efficiency over a year for each studied packet type

Packet. So if an improvement in throughput is more important than a high compression ratio, *Unified* or *Packet* is the best approach.

6.3 Normalized Compression Energy

Figure 5 shows the energy efficiency of each of the compression approaches relative to the baseline in megabytes per joule. We use this metric because it is independent of the packet size. Again, the *y*-axis is logarithmic and higher numbers are better.

We find that the general trend of the graphs is the same as in Figure 4, i.e., runtime and energy are correlated. While *Unified* and *Packet* behave roughly the same in terms of energy as they do in terms of throughput (relative to the baseline), *Adaptive* does better, meaning that *Adaptive* runs at a lower average wattage than the baseline, but *Unified* and *Packet* run at about the same wattage as the baseline. Conversely, gzip and bzip2 are much worse and draw much more power than the baseline. This is expected as gzip and bzip2 use much more memory and comprise more complicated algorithms than the baseline or our custom algorithms.

Unified is best in terms of consuming the least amount of energy to compress data, followed closely by *Packet*. However, if the goal of lowering the energy is to reduce the power draw (i.e., the wattage), *Adaptive* is better than any of the other approaches including the baseline.

7. CONCLUSIONS & FUTURE WORK

In this paper, we study a method of generating fixed and adaptive compression algorithms for use on spacecraft, where hardware limitations restrict the complexity and memory usage of these algorithms. For this purpose, we wrote a tool called SDcrush that automatically synthesizes compression algorithms from training data.

By breaking existing compression algorithms down into their constituent algorithmic components, rejecting those components that are too complex, too slow, and/or use too much memory, we derive a set of components that can be chained together to form custom compression algorithms that are tuned to perform well on a specific data packet. We study the effects of using one fixed algorithm for an entire spacecraft (*Unified*), one fixed algorithm per data packet type (*Packet*), and using an algorithm that is adaptive based on recently received data (*Adaptive*). We compare our custom approaches to the baseline (the algorithms THEMIS B is currently using) as well as to commonly used compression algorithms.

We found that our *Unified* approach performs about twice as well as the baseline in terms of throughput and energy consumption but performs the worst out of all our studied approaches in terms of compression ratio.

By adding a layer of specialization with our *Packet* approach, we improve the overall compression ratio over *Unified* by 9% to 45%, depending on the packet type. However, doing so sacrifices throughput by between 10% to 40% and similar amounts of energy.

Adaptive performs on par with *Packet* in terms of compression ratio but is much worse for energy and especially throughput. Interestingly, *Adaptive* runs at a lower wattage than any of the other approaches. The main strength of *Adaptive* lies in its ability to be reconfigured to adjust to changing space conditions, though.

The general-purpose algorithms gzip and bzip2 deliver the highest compression ratios but perform poorly in terms of throughput and especially energy. Furthermore, their advantage in compression ratio is in part due to their large memory usage, which greatly exceeds the amount of memory available on THEMIS B.

We conclude that significant improvements in compression ratio, throughput, and energy can be obtained by generating custom compression algorithms from sets of chainable compression components.

SDcrush is currently using algorithm components that we extracted from floating-point and trace compressors, i.e., domains that are very different from space data. In the future, we plan to include components that are specifically tailored to space data as well as instrument-specific components, which should boost the compression ratio and may make compression even faster.

We also plan to generate custom compression algorithms using data from THEMIS C as it resides in a similar plasma environment as the probe used in this study. This will give us insight into how the algorithms may differ between probes in the same space environment. Similarly, we plan to synthesize custom algorithms for THEMIS A, D, and E, which reside in fairly different plasma environments than B and C. We expect their algorithms to be quite different, which should shed light on how important adaptivity is.

To address the problem of *Adaptive* negatively affecting compression ratios over *Packet*, we plan to develop a method for eliminating the "rebound" that appears after a highly effective algorithm has been used on an atypical data packet. One such method would be to design a doubly adaptive approach that also varies the number of prior days that are considered when synthesizing the next algorithm to use.

ACKNOWLEDGMENTS

Jared and Martin are supported by the U.S. National Science Foundation under grants 1217231, 1406304, and 1438963 as well as a REP grant from Texas State University. Andrew acknowledges support from NASA's SSERVI/DREAM2 Institute, grant #NNX09AI36A. He also thanks Dr. Lewis for assistance in obtaining and interpreting the THEMIS on-board compression algorithms. The ARTEMIS mission is funded and operated under NASA Grant #NAS5-02099 and we acknowledge Dr. Angelopoulos for the use of THEMIS/ARTEMIS mission data. All ARTEMIS data used in this work are publicly available at http://artemis.ssl.berkeley.edu.

REFERENCES

[1] J. Ziv and A. Lempel. "A Universal Algorithm for Sequential Data Compression." *IEEE Transactions on Information Theory*. May 1977

[2] J. Plaunt and R. Rice. "Adaptive Variable-Length encoding for Efficient Compression of Spacecraft Television Data." *IEEE Transactions on Communication Technology*, vol. COM-19. 1971.

[3] P. Yeh, W. Miller, and R. Rice. "Algorithms for a Very High Speed Universal Noiseless Coding Module." JPL Publication 91-1, 1991.

[4] P. Yeh. "Implementation of CCSDS Lossless Data Compression for Space and Data Archive Applications." *CCSDS Space Operations Conference*. 2002.

[5] "Lossless Data Compression: Recommendation for Space Data Systems Standards." *CCSDS 121.0-B-1*. Issue 1. May 1997.

[6] "Lossless Data Compression: Green Book." *CCSDS 121.0-G-1*. Issue 1. May 1997.

[7] J. Portell, G. Artigues, R. Iudica, and E Garcia-Berro. "FAPEC-based lossless and lossy hyperspectral data compression." *High-Performance Computing in Remote Sensing V*. October 2015.

[8] S. Bhattacharya, T. Blumensath, B. Mulgrew, and M. Davis. "Fast Encoding of Synthetic Aperture Radar Raw Data Using Compressed Sensing." *Statistical Signal Processing*. 2007.

[9] A. Barrie, M. Adrian, P. Yeh, G. Winkert, J. Lobell, A. Vinas, and D. Simpson. "Fast Plasma Instrument for MMS: Data Compression Simulation Results." *American Geophysical Union*. 2009.

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

[11] M. Burtscher, 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*. November 2016.

[12] V. Angelopoulos. "The THEMIS Mission." *Space Sci. Rev*., 141, 5-34. 2008.

[13] V. Angelopoulos. "The ARTEMIS Mission." *Space Sci. Rev*., 165(1-4), 3-25. 2011.

[14] E. Taylor, P. Harvey, M. Ludlam, P. Berg, R. Abiad, and D. Gordon. "Instrument Data Processing Unit for THEMIS." *Space Sci. Rev*., 141, 153-169. 2008.

[15] gzip: http://www.gzip.org/

[16] bzip2: http://www.bzip.org/

[17] D. Huffman. "A Method for the Construction of Minimum-Redundancy Codes." *Proceedings of the IRE*. 1952.

[18] M. Burrows and D. Wheeler. "A Block Sorting Lossless Data Compression Algorithm." *Technical Report 124, Digital Equipment Corporation*. 1994.

[19] PAPI: http://icl.cs.utk.edu/papi/