UNIVERSITY OF CALIFORNIA

Los Angeles

Datacomp:
Locally-independent Adaptive Compression for Real-World Systems

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Peter Andrew Harrington Peterson

2013

ABSTRACT OF THE DISSERTATION

Datacomp: Locally-independent Adaptive Compression for Real-World Systems

by

Peter Andrew Harrington Peterson

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2013

Professor Todd Millstein, Co-chair

Professor Peter Reiher, Co-chair

Typically used to save space, non-lossy data compression can save time and energy during communication if the cost to compress and send data is less than the cost of sending uncompressed data. However, compression can degrade efficiency if it compresses insufficiently or delays the operation significantly, which can depend on many factors. Because predicting the best strategy is risky and difficult, compression (if available) is typically manually controlled, resulting in missed opportunities and avoidable losses.

This dissertation describes Datacomp, a general-purpose Adaptive Compression (AC) framework that improves efficiency in terms of time, space and energy for real-world workloads on real-world systems like laptops and smartphones. Prior systems are limited in important ways or rely on external hosts for prediction and compression, reducing their effectiveness or imposing unnecessary dependencies. In contrast, Datacomp is a Local Adaptive Compression system capable of choosing between numerous compressors using system monitors, a novel compressibility estimation technique and a history mechanism. Datacomp wraps system calls with AC capabilities, enabling applications to benefit with little modification. I also built Comptool, an off-line “AC oracle” for investigation and validation. Comptool, which includes LEAP energy-measurement capabilities, identifies the best-case compression strategy for a given scenario, highlighting critical factors for AC and providing a valuable standard against which to compare systems such as Datacomp.

I evaluated two Datacomp-enabled utilities: drcp, a throughput-sensitive remote copy tool and dzip, an AC-enabled compression utility. I collected hundreds of megabytes of nine common but distinct classes of data to serve as workloads, including web traces, binaries, email and collections of personal data from volunteers. Experiments were performed using both Comptool and Datacomp while varying the data type, bandwidth, CPU load, frequency, and more. Up to and including 100Mbit/s, Datacomp consistently came within 1-3% of the best strategy identified by Comptool, improving throughput for realistic types by up to 74% over no compression, and up to 45% over zlib compression. Comptool generated strategies that could improve efficiency at gigabit speeds (over no compression) by up to 28% for Wikipedia data and 14% for Facebook data.

The dissertation of Peter Andrew Harrington Peterson is approved.

William Kaiser

Douglas Stott Parker

Junghoo Cho

Peter Reiher, Committee Co-chair

Todd Millstein, Committee Co-chair

University of California, Los Angeles

2013

DEDICATION

"Bernard of Chartres used to say that we are like [puny] dwarfs on the shoulders of giants, so that we can see more than them, and things at a greater distance, not by virtue of any sharpness of sight on our part, or any physical distinction, but because we are carried high and raised up by their giant size."
– John of Salisbury, Metalogicon [1] (1159)

I like this version of the famous thought because of its humility. Seeing farther by virtue of standing on giants' shoulders doesn't require one to be a giant themselves, or even to see especially far. It merely requires a person who is willing to look and some patient, generous and steady giants willing to lift. I have been blessed with many such giants.

First, I would not have completed this process without the love of my life, Anna. She quite literally supported this endeavor in every way as it evolved from a two-year Master's degree into something much, much larger. She has been incredibly patient and gracious and has helped me grow through this process in many ways that are more personally valuable to me than the degree.

Thank you to my parents, Tom and Sue, for everything, including instilling in me a love of learning, creating, teaching and experimentation. Dad, if you hadn't brought those TRS-80s and Apple][s home from the school district over all those summer vacations I would never have ended up here. Mom, thank you for your attention to detail, which has served me well in writing both text and code. (I've hidden some typos in here for you to find.) I thank the Harringtons and the rest of my family for their love and support. Thank you to everyone who gave me the benefit of the doubt during this project. I've bit off more than I could chew in the past, but this was something else entirely. I owe you.

I also would never have done any of this if not for the support and encouragement of Dr. Peter Reiher, who gave me the chance to seek this Ph.D. His practical advice and the environment in his lab defined my UCLA experience. Thank you as well to Janice for her support and sharp proofreading, to my lab mates and colleagues in the CSD, and to the giants listed in Section 14.

Finally, many people, perhaps unknowingly, made various contributions to this project: Dr. William Kaiser and Digvijay Singh, Dr. Junghoo Cho, Dr. D. Stott Parker, Dr. Todd Millstein, Dr. Jelena Mirkovic, Dr. Tanya Crenshaw, Dr. Eddie Kohler, Dr. Paul Eggert, Dr. Alan Iliff, Dr. Alice Iverson, Dr. Joe Lill, Dr. Walter M. Gibbs, Dr. Michael Meisel, Dr. Erik Kline, Dr. Alex Afanasyev, Dr. Chuck Fleming, Vahab Pournaghshband, Matt Beaumont-Gay, Elizabeth Harrington, P. Joshua Griffin, Lukas Eklund, Jess Frykholm, Max Peterson, Clint and Charles Bergsten, James Herrick, Eric and Robin Berglund, Louise Ambros, and the "usual gang of idiots," including Nick Moffitt, Paul Collins, Brian Hicks, Emad El-Haraty, Neale Pickett and Ryan Finnie. Thanks and apologies to those inexplicably omitted.

I have had the privilege to stand on the shoulders of a great group of wonderful, friendly and brilliant people. This dissertation is dedicated to you.

Table of Contents

List of Figures

List of Tables

Acknowledgments

This work was supported under NSF grant CNS-1116898.

The author also gratefully acknowledges the support of the Intel Corporation for certain portions of this work.

Vita

1999	B.M.E. (Bachelor of Music Education). North Park University, Chicago, Illinois
1999-2005	PC Coordinator. North Park University, Chicago, Illinois
2003	Instructor, Computer Networking. North Park University, Chicago, Illinois
2005-2006	Director of Information Technology, The Yucaipa Companies. West Hollywood, California
2007-2008	Teaching Assistant, Computer Organization, Undergraduate Computer Security and Graduate Computer Security (Online MS Program). Computer Science Department, UCLA
2008-present	Research Assistant, Laboratory for Advanced Systems Research. Computer Science Department, UCLA
2009	M.S. Computer Science. UCLA, Los Angeles, California
2010-2011	Teaching Assistant, Undergraduate Computer Security and Graduate Computer Security (Online MS Program). Computer Science Department, UCLA
2011	Co-Instructor. Embedded Systems Security. Computer Science Department, UCLA

PUBLICATIONS

P. A. H. Peterson and P. Reiher. Security Exercises for the Online Classroom with DETER. In the 3rd USENIX Workshop on Cyber Security Experimentation and Test, August 2010.

P. A. H. Peterson. Cryptkeeper: Improving Security with Encrypted RAM. In Proceedings of the IEEE Conference on Technologies for Homeland Security (HST), November 2010.

D. Singh, P. A. H. Peterson, P. Reiher and W. Kaiser. "The Atom LEAP Platform For Energy-Efficient Embedded Computing: Architecture, Operation, and System Implementation", December 2010.

P. A. H. Peterson, D. Singh, W. Kaiser and P. Reiher. Investigating Energy and Security Trade-offs in the Classroom with the Atom LEAP Testbed. In the 4th USENIX Workshop on Cyber Security Experimentation and Test (CSET), August 2011.

J. Mirkovic, M. Ryan, J. Hickey, K. Sklower, P. Reiher, P. A. H. Peterson, B. H. Kang, M. C. Chuah, D. Massey and G. Ragusa. Teaching Security with Network Testbeds. In the Proceedings of the ACM SIGCOMM Workshop on Education, August, 2011.

A. Fujimoto, P. A. H. Peterson and P. Reiher. Investigating the Energy Costs of Full Disk Encryption. In the Workshop on Energy Consumption and Reliability of Storage Systems (ERSS), part of the 2012 International Green Computing Conference (IGCC), June 2012.

C. Fleming, P. A. H. Peterson, E. Kline and P. Reiher. Data Tethers: Preventing Information Leakage by Enforcing Environmental Data Access Policies. In Proceedings of the IEEE International Conference on Communications (ICC), June 2012.

M. Gray, P. A. H. Peterson and P. Reiher. Scaling Down Off-The-Shelf Data Compression: Backwards-Compatible Fine-Grain Mixing. In Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS), June 2012.

1 INTRODUCTION

"Compression is desirable on modem lines and other slow connections, but will only slow down things on fast networks." – The OpenSSH manual [2]

One major component of the value of computers is their ability to perform information processing tasks very quickly. For this reason, efficient use of resources has been critical since the very early days of the field. Historically, these resources have been storage space and computation time – the "matter and energy" of the information universe. Recently, however, the technology community has also begun to consider efficient use of energy as a critical goal due to hardware evolution issues, the expansion of battery powered mobile devices, and the increasing economic and environmental costs of computation.

Many advances in theory, hardware, and software begin as attempts to improve performance through more efficient use of resources. Sometimes, promising ideas are sometimes dismissed outright, are overlooked, or languish unused for years despite their potential benefits because in practice they are too computationally demanding for the hardware of the era. Or, as was the case with parallel computing until recently, a capability may be too economically expensive for widespread use in general computing systems. Sometimes, technologies that could improve performance are underutilized because they are misunderstood or because they are difficult to manage. Non-lossy compression[1] is one such technology.

Non-lossy compression describes a class of algorithms that improve spatial efficiency by "spending" computation time in a search for more efficient representations of information. Compression algorithms take an input and attempt to produce an output that uses fewer bytes but

[1] This work is almost exclusively concerned with non-lossy compression techniques, where the compression operation is a reversible transformation. This is contrasted with lossy compression, also known as "perceptual encoding", where space is saved by reducing the fidelity of the data considered less important while retaining higher fidelity for critical data. See Section 2.

that can be reversibly “decompressed” into the original input. While compression is fundamentally about reducing the number of bits required to express information, it can also save time. Because compression can reduce the number of bits required to represent information, it can allow the same information to be transferred between two parties in less time.

Surprisingly, even though compression always requires computation, it can often be used to save a significant amount of energy. Approximately 1,000 operations can be performed on a bit for the same amount of energy spent to transmit that bit over a network [3]. As a result, if the cost-benefit ratio is favorable, compression, much like an efficiency expert for hire, can save more resources than it costs, resulting in a net gain. Unfortunately, compression is not always helpful. In fact, when it is not able to save space, it is actively detrimental to efficiency because the effort spent in the attempt to compress produces nothing of value. What’s more, many combinations of algorithms and data can actually be pathological, outputting “compressed” data that is larger than the original, wasting space in addition to time and energy.

It would be convenient if, for any given scenario, we could accurately predict whether compression would be worthwhile or wasteful. With that ability, we could always make the right choice, reaping the benefits of compression when possible, and paying nothing (or very little) when compression was not worthwhile. Even if we could only make the right prediction *most of the time* we could improve efficiency in many cases. Unfortunately, making these predictions accurately and efficiently is not a straightforward task. Choosing the best compression strategy for a given opportunity is difficult because many factors affect whether, and how, efficiency can be improved through compression. These include factors related to the algorithm choice and configuration, data size and characteristics, execution environment properties, and practical considerations related to application constraints. Making the perfect choice often requires information that is not available when the decision must be made.

As a result of this difficulty, most applications, if they do so at all, support compression using a manual, “all or nothing” approach. They typically support only one compression technique, often chosen not to maximize efficiency for the application in question, but based on popularity or intuition. Even in these cases, compression is typically not enabled by default because it will degrade performance in certain circumstances. Unfortunately, once compression is enabled it is typically applied to *all data*, regardless of the value of the particular operation. Examples of these systems include file systems that are capable of performing compression and the networking tool OpenSSH. The optimal application of compression requires intelligence – but in practice (aside from initial user choices or very simple systems), there is usually very little intelligence involved.

However, new developments in technology and industry have created potential for change. The importance of efficiency has been making an impact at all levels. Data centers around the world consume an enormous amount of electricity in the service of powering and cooling computer systems; reducing this consumption by even a single percent could have a significant impact. At the other end of the spectrum is the explosion of mobile devices relying on limited batteries and shared communication channels. The limited computing power of mobile devices like smartphones in combination with market forces has driven corporations to investigate using compression in new ways to improve the performance of their products. At the same time, the expansion of high-performance distributed systems such as grid, cluster, and cloud computing have driven research to improve the efficiency of communication between nodes via overburdened IO channels. In other words, the value of improving efficiency has perhaps never been greater.

Nascent techniques for automatically improving efficiency through compression are being or have been developed, such as techniques that can automatically vary compression strategy based on system conditions and methods to efficiently predict how well a particular input will compress. Other research has focused on leveraging third party systems to improve the efficiency of specific client machines. Developments like these have opened the door for broader, automated, and more intelligent use of compression in software systems. This relatively new field, called "Adaptive Compression," attempts to develop and leverage compression estimation and application techniques to automatically monitor and apply compression whenever profitable for saving time, space and energy.[2] Additionally, new developments have made compression more useful than ever: compression research has developed high-speed algorithms that can be profitable in formerly hopeless scenarios, multi-core hardware and software parallelism can be used to meaningfully improve compression throughput in many cases, and new techniques for applying existing algorithms have made them more flexible than in the past.

Adaptive Compression (AC) has been attracting interest, especially in high-performance distributed computing (HPDC) and certain types of mobile data optimization. HPDC is an interesting venue for AC, because performance is vital, workloads may be predictable, there is can be a central control or dispatch mechanism with a global view, middleware supporting the distributed system can often have compression added to it (rather than the OS or applications), and adding hardware to help make compression decisions is trivial ("What's one more

[2] Somewhat confusingly, "Adaptive Compression" also refers to a common and much lower-level compression technique wherein algorithms modify the expected probabilities of symbol appearance while compressing.

machine?”). Thus, AC solutions for HPDC can rely on computationally expensive or exotic solutions as long as they improve performance.

For different reasons, Adaptive Compression can also be attractive for mobile device vendors. Mobile devices are typically battery-powered and have low computational power compared to more typical personal computers. They also rely on the public cellular network, a shared wireless medium which, compared to wired Internet service, is slower and more monetarily expensive on a per-byte basis[3]. It is a real value for users to transmit and receive fewer bytes, from the perspective of battery life and run time – not to mention dollars and cents. Motivated to sell their products, but ostensibly constrained by the low computational and network resources of these devices, vendors such as Google, Nokia and Opera Software (makers of the Opera browser) have developed proxy-based solutions [4] [5] [6]. In these solutions, the products are engineered to trust the vendors, who provide remote proxy servers to compress data on behalf of the end users. Nokia’s product even trusted the proxy to transparently decrypt and re-encrypt HTTPS sessions – a major security issue.

While the limited solutions used for HPDC and mobile devices can improve efficiency in these environments (and for certain definitions of efficiency), they are not practical for the general computing world. In the HPDC environment, extra hardware or AC support mechanisms are essentially part of the maintained system even if they are not part of the compute nodes. In the mobile world driven by hardware and software vendors, AC based on remote support (such as a proxy) is an application purchased along with a product (the phone or browser) and is subject to all the typical limitations of a software service. These approaches are open to criticism

[3] As of April 26, 2013, Verizon charges $15 for each gigabyte of data over a user’s contract plan (which is normally in the low single digit gigabytes per month), as opposed to most home Internet services which charge $25-50 per month for unlimited access at much higher speeds.

for a variety of reasons discussed later in this work. But above all, they are a dependency on systems that reduces the potential impact of AC because they require external hardware to exist and network connectivity to be available. Users without access to (or with reservations about using) a proxy or computers that are not always connected to the network (but that might be able to benefit from AC internally) have no such remote hosts available and so cannot benefit from these kinds of AC approaches.

This raises a question: rather than requiring dedicated hardware and being used in specialized environments or applications, is it possible for AC to be profitable for generic computer systems processing arbitrary, user-generated workloads while relying only on local knowledge and resources? We might call this system "AC for the PC." Such a system could be tailored to its local hardware and workloads.

If successful, it would be able to extract additional performance and value from existing hardware that is currently being wasted. In the worst (successful) case, such a system would "break even" or not improve efficiency significantly. In the best case, it could be useful enough to become a feature of the operating system – improving efficiency in general and removing from users the burdens and risks inherent in making compression choices by hand. An operating system with integrated AC might also be able to improve the efficiency of communication operations beyond basic networking, such as improving the efficiency of storage and memory.

This dissertation describes the research and development surrounding the construction of one such locally-independent Adaptive Compression system and related tools. The project, "Datacomp," has produced multiple pieces of work, including: an investigation into non-lossy compression and the factors affecting its use for improving efficiency; a common terminology for AC systems and components; Comptool, an offline tool capable of identifying the best (or close to the best) strategies for a wide range of scenarios; libdatacomp, a general-purpose adaptive compression framework designed to transparently optimize compression decisions in real time using local information; a mechanism for efficiently estimating the compression properties of an input; and two applications demonstrating the use and effect of the framework.

In addition to not using remote hosts to facilitate adaptation, Datacomp improves upon the limited research in adaptive compression by being more general and modular in many ways. For example, it does not assume a specific application area, amount or diversity of input, set of compression algorithms or compression parameters. Nor does Datacomp assume a specific optimization target (e.g., time), but rather is able to make decisions based on a user-determined priorities for time, space, and energy resources. Datacomp is also aware of and leverages modern advances in hardware such as frequency scaling and parallelism.

Evaluating a system like Datacomp is challenging for several reasons. First and foremost, while we can identify relative improvement by comparing Datacomp's results against simple, static compression strategies such as "no compression" or compression with specific algorithms, we would ultimately like to know how much *room for improvement* exists in a given application area. If it is impossible for Datacomp to significantly improve efficiency, perhaps it is not worth going to the trouble. On the other hand, if there is significant room for improvement, but Datacomp is unable to obtain that benefit, then we know that more development or research is needed to improve efficiency or explain why it cannot be improved.

Additionally, targeting "the general computing environment" is a challenging goal. The types of applications and data that users process are numerous, varied, and known only in a general sense. Attempting to create a compression corpus of "typical user data" is a significant research challenge in and of itself. Rather than attempting to test every possible type of scenario (a fool's errand), an evaluation for Datacomp must include enough types of data and different circumstances to compellingly suggest that Datacomp is likely to be successful in most best, worst, and common cases. To achieve this, we have collected a wide range of data from volunteers, including file and web data in controlled and uncontrolled tests as well as theoretical best and worst case workloads. In the evaluation, we test Datacomp's effectiveness on this data in a wide variety of execution environments including various network bandwidths and CPU loads.

Ultimately, the value of Datacomp as a project does not hinge simply on the performance of libdatacomp. While libdatacomp's performance is important and is perhaps the best and most obvious evidence of the value of this work, the long-term value of this project is its attempt to "crack open" the problem of profitably performing Adaptive Compression on local systems. By identifying and working through the challenges to Local Adaptive Compression in real-world systems, Datacomp provides a first step towards realizing the benefits of Adaptive Compression in general-purpose computing devices.

2 Non-Lossy Compression

Before discussing related work and digging deeper into the particulars of Datacomp and Comptool, it is helpful to have an informal grounding in how non-lossy compression works and how it can be used to improve efficiency in areas other than reducing storage space. It is also important to understand the factors that affect compression outcomes so as to better understand how making profitable compression choices on the fly is challenging but nevertheless possible. This section presents an informal introduction to non-lossy compression, highlighting these issues and building a foundation for the rest of this work.

2.1 Basic Techniques, Compression Tools, and Options

A non-lossy compression algorithm is a computational method by which symbolic information can be analyzed and re-encoded so that it expresses exactly the same information in a more compact form. Compression is often successful on human-generated symbolic information (including not only language but computer code and structured data) because most meaningful data incorporates a significant amount of redundancy in terms of repeated patterns and symbols. Compression saves space by reducing this redundancy in a reversible way, while decompression recovers the original data.[4]

When one hears the term "compression algorithm," names like "gzip," "bzip2," "LZO," and "PKZIP" come to mind. These are the compression tools or utilities that people use and which are themselves made of one or more fundamental compression techniques combined with input and output routines, checksums, and so on. Compression utilities typically include

[4] The obvious example of "meaningful data" that does not compress well with non-lossy techniques is multimedia, where the inherent "structure" that provides meaning is not expressed symbolically but rather in terms of digitized patterns that are interpreted visually or aurally. While many of the techniques discussed in this document could work for lossy compression (and while I test Datacomp on hard-to-compress multimedia data), here "compression" refers to non-lossy compression exclusively.

decompression routines in the same executable or in a different utility that is included in the installation package.

While there are many distinct compression utilities, there are only a handful of fundamental compression techniques, which are combined into the higher-level algorithms or utilities that users apply. But the algorithm alone does not determine the performance of a particular compression operation. Rather, compression performance (in terms of data reduction and run time) can vary widely based on interacting combinations of method, software implementation, input characteristics, software execution environment, hardware characteristics, and more. Because of the importance of performance, most compression implementations support a variety of run-time options determining how memory and computational resources should be used during compression.

Because of these complex interacting issues, it is often easy to make a good choice for compression, but it can be difficult to make the best choice. Making the best choice possible is the core goal of Adaptive Compression. Thus, in order to set the stage, the following sections describe some of the concepts behind non-lossy compression including issues affecting compression performance. I first give an introduction to the basic ideas and nuances of compression, followed by a description of some of the most widely-used real-world techniques. Finally, various applications of compression face unique challenges due to practical constraints, which can affect how compression systems are designed in terms of both algorithm selection and architecture. We close this section with a discussion of some of those issues.

2.2 A Compression Primer

Before delving into the real-world mechanisms and designs, is helpful to have an intuitive understanding of how compression works and why making the best possible compression choices is a challenging task.

The process of compression involves analyzing a stream of symbols (i.e., bits or bytes) in an attempt to find a way to represent the same information using fewer symbols. The classic approach uses a two-step process where a statistical model of the data is made first, after which the data is encoded by describing exceptions where the real data deviates from the model. Of course, both the model and the list of exceptions require space to describe, so a key goal is expressing both in an optimally efficient manner. In the end, if the model and the deviations from the model require fewer bits to express than the original stream, then compression is successful.

2.2.1 EngZip

With compression, the goal is to make the most useful model possible within some constraint (such as a limited amount of working memory) while also minimizing the size of the output and the required computation time. The design of a multi-purpose algorithm must strike a balance between maintaining the power to compress general input while also maximizing the effect and efficiency of compressing the specific input in question. General approaches may work well overall, but they may miss opportunities for better performance. On the other hand, more aggressive approaches can be costly and complicated. Additionally, model choice matters (and would ideally be based on the input), and there are tensions between achieved compression and the complexity, time and memory costs of the method. In this section, we explore these and other compression issues using EngZip, a hypothetical "mental compression algorithm."

EngZip's compressed format is composed of complete sentences of English prose describing the input. For the sake of argument, suppose that in EngZip, encoded data must be *described* in natural language – we cannot quote more than one letter at a time. To decompress data compressed with EngZip, we simply use our minds to interpret the prose and reconstruct the original text. (For the sake of argument, we assume that the descriptions are unambiguous.)

Data Type	Content	Total Bytes	CR
ASCII	ABABABABABABABABABABABABABABABABABABABABABABA (...one megabyte later...) EOF	1,045,876	1
EngZip	one megabyte of alternating As and Bs but where the last three bytes are E, O and F	84	.00008

Table 1. EngZip achieves an incredible compression ratio of 84:1,045,876 or .00008!

Suppose we want to use EngZip to compress a one-megabyte stream composed of alternating As and Bs, but where the last three bytes are E, O and F. Actually, the previous sentence contains a valid EngZip-compressed version of that stream. The first part of the sentence describes the model –"one megabyte of alternating As and Bs" –while the last clause describes the exceptions to the model –"but where the last three bytes are E, O and F." This "compressed" version, shown in Table 1, is only 84 bytes long, versus the original of one megabyte (1,048,576 bytes). Thus, we say that EngZip, with the given input, achieves the compression ratio (CR) of 84:1,048,576, or .00008. EngZip is therefore an excellent compression algorithm for this input, since most real-world compressors only achieve a CR of between 0.2 and 0.8 for compressible data, with 0.5 (50%) being a middle-of-the-road value.

2.2.2 Variation by Input and Algorithm

EngZip will obviously fail miserably for streams that are necessarily shorter than their shortest descriptions or are otherwise difficult to describe in English. For example, suppose we wanted to compress the stream "abcdefghijklmnopqrstuvwxyz". A valid EngZip-compressed version is the string “It is the lowercase alphabet”, shown in Table 2. This string is 28 characters, although the lowercase alphabet itself is only 26 characters, for a CR of 28:26, or 1.03. The compressed stream is actually larger than the original stream.[5] Even worse, consider the outcome of EngZip-compressing a single symbol; “A” results in the output “It is the letter A” – which is eighteen times larger than the original.

Data type	Content	Total Bytes	CR
ASCII	abcdefghijklmnopqrstuvwxyz	26	1
EngZip	It is the lowercase alphabet	28	1.03

Table 2. The Alphabet compressed with EngZip actually expands -- with a compression ratio of 28:26 or 1.03.

While the mechanics are different in practice, these examples demonstrate how, due to the assumptions inherent in any given algorithm, different combinations of algorithms and input can result in widely varying outcomes. Thus, when compression ratio is of the utmost importance, choosing the right algorithm can make a significant difference. While some algorithms are just generally more effective than others, individual algorithms can also have poor performance on particular kinds of data, regardless of length.[6] Conversely, specialized algorithms can prove highly effective when applied to certain kinds of input – we see this in EngZip with its talent for compressing the example in Table 1. A real-world example of this kind of special-purpose algorithm is the Free Lossless Audio Codec [7] (FLAC), a non-lossy compression algorithm designed specifically for PCM audio (.WAV files).

2.2.3 Variation in Compressibility

Some data is simply not very compressible because it does not have much redundancy. In general, the less structure or repetition present in the stream, the less effective compression will

[5] One might be tempted to try “lowercase alphabet” as the compressed version, but that is not a complete sentence (although it might be valid SlangZip output).

[6] As a silly example, EngZip would probably not achieve a very good average compression ratio for text written in Chinese, because each Chinese character would require many English characters. For the same reason, EngZip is not very good at compressing numerical data.

be. While many files we encounter in the real world are compressible, from an information theory perspective compressible strings are the exception, not the rule. In fact, we can easily prove that most arbitrary streams of a given length cannot be reversibly compressed to some reasonable degree (e.g., 50%). There are 4,294,967,296 unique 32-bit strings, which, if they could all be reversibly compressed to 50%, would require only 16 bits each for storage. But of course there are only 32,768 unique 16-bit strings! Using the 32,768 16-bit strings as the compressed versions of the four billion 32-bit strings results in 131,072 collisions for every 16-bit string. Thus, at most only 32,768 32-bit strings can be reversibly compressed to 16 bits. This limitation applies to any reasonable compression ratio, not just 50%, because compressing all *n* bit strings to *(n-1)* bit strings means a halving of the number of unique possible strings. Thus, the vast majority of arbitrary n-bit streams cannot be reversibly compressed to some arbitrary size [7].

Common examples of non-multimedia real-world hard-to-compress data include that which is random or approaches statistical randomness, such as encrypted or previously compressed data. This kind of data is difficult to compress because most of the redundant structure has already been eliminated (with compression) or hidden (with encryption). For these types of data, any compression at best wastes the energy required to compress the data, and at worst can expand the output (worse than a waste), because the “compressed” version of the stream includes extra symbols necessary for decompression.

For example, Table 3 shows that EngZip would perform badly when faced with a random permutation of the alphabet because there is no alternative but to “spell it out” letter by letter. Allowing verbatim quotation – a feature supported in many compressors and proposed for inclusion in EngZip 2.0 – would allow the compressor to simply quote the raw text when it recognizes that compressing the data would be unprofitable. Still, as shown in Table 3, indicating that the compressed data is a quotation still costs two bytes – in this case, a pair of quotation marks.

Data type	Content	Total Bytes	CR
ASCII	Pszieblfvgdxhqjawkmontyurc	26	1
EngZip	It is the string P, S, Z, I, E, B, L ...	97	3.73
EngZip 2.0	“pszieblfvgdxhqjawkmontyurc”	28	1.07

Table 3. Statistically random data is typically hard to compress.

2.2.4 Input Length and Compression Ratio

Somewhat counter-intuitively, longer inputs generally result in better *compression ratios*, because repetitions and other structure in a longer stream may repeat and thus comprise a larger portion of the input. Suppose that, rather than “abcdefghijklmnopqrstuvwxyz”, our input string is three repetitions of the lowercase alphabet, totaling 78 characters. Table 4 shows us that “It is the lowercase alphabet, three times” is 41 characters, resulting in a CR of 41:78 or .526 as opposed to a CR of 1.03 for the single alphabet in Section 2.2.2. The improvement grows with the duplications because the basic method for representing one repetition can be reused quite cheaply for many more, as shown in Table 5; “It is the lowercase alphabet, quadrupled” is even better – 40 characters, for a ratio of 40:104, or 0.384.

However, more isn’t always better. Reading and processing longer inputs takes more time, which can delay output depending on how much data must be read before output can be generated by the particular compressor in use. Larger inputs can also require using more memory, depending on the algorithm. Finally, there is a tension between compressing large portions of input with one algorithm (to improve compression), and compressing smaller portions of input in sequence (which provides opportunities to fine-tune a compression strategy).

Data type	Content	Total bytes	CR
ASCII	abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz	78	1
EngZip	It is the lowercase alphabet, three times	41	0.526

Table 4. Longer inputs often enable better compression. EngZip achieves 41:78 or 0.526 on this string.

Data type	Content	Total bytes	CR
ASCII	Abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxy zabcdefghijklmnopqrstuvwxyz	104	1
EngZip	It is the lowercase alphabet, quadrupled	40	0.38 4

Table 5. EngZip achieves 40:104 or 0.384 on this string.

2.2.5 Compression Blocks and Sliding Windows

In practice, compression algorithms require differing amounts of resources based on the underlying techniques being used and the implementation. Memory can improve compression ratio by allowing the algorithm to analyze larger portions of the input stream at once. While gzip [8] compresses data as a byte stream, it maintains its compression history in a sliding window of 32KB. Other algorithms, such as bzip2 [9], read and compress the input stream in fixed-size blocks. These design choices are made for various reasons, such as to limit the amount of memory required at once. However, they can also affect the achievable compression.

In our first example (Table 1), EngZip could not have achieved such effective compression if it hadn't read the entire one megabyte of alternating As and Bs before generating output. Table 6 shows how if EngZip was limited to compressing input in 52 character blocks, the "quadrupled alphabet" input (104 characters) could not be compressed as one block – it would have to be divided and then compressed, resulting in a meaningful loss of compression. Options like the amount of memory to use or the window size are often user configurable (especially at the library level), but most users use the defaults simply because the best choice is not known and difficult to determine.

Data type	Content	Total bytes
ASCII	Abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd efghijklmnopqrstuvwxyz	104
EngZip	It is the lowercase alphabet, doubled; It is the lowercase alphabet, doubled	75

Table 6. A block size of 52 characters reduces EngZip's ability to fully leverage the redundancy. Since it can only consider two repetitions of the alphabet at a time, it achieves a ratio of only 75:104 or 0.721.

2.2.6 Throwing Computation at the Problem

More computation also usually results in better compression, to a point. Compression can be described as a search for more efficient representations. Thus, more cycles spent searching (for example, using multiple passes or backtracking) can sometimes find more compact encodings for a given stream. Of course, more computationally expensive algorithms require more time and energy, which can create delay and affect the system in undesirable ways. In order to provide users with flexibility, many algorithms provide a “strength” parameter to control the amount of resources expended in the service of compression.

To extend the EngZip analogy (perhaps to the breaking point), suppose that EngZip’s “strength” setting determines the maximum number of seconds that can be spent thinking of shorter, equivalent sentences. When the time limit is up, the current best representation is used or the string is spelled out verbatim as in Table 3 if no compression was achieved. Table 7 shows the effect of using more “computation”; after some pondering we realize that by using the word “thrice” instead of the phrase “three times”, we can eliminate 6 letters, improving the compression ratio to 34:78 or 0.436 for a savings of 21% (over 0.526 the ratio achieved in Section 2.2.4).

Data type	Content	Total bytes	CR
ASCII	abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz	78	1
EngZip	It is the lowercase alphabet, thrice	34	0.436

Table 7. A little extra computation reveals that using "thrice" instead of "three times" improves EngZip's compression ratio for this string by 21%.

2.3 Fundamental Compression Mechanisms

Now that we have discussed how compression works in general, along with some of the trade-offs inherent in compression approaches, it is worthwhile to describe some of the real-world low-level compression mechanisms in use. The mechanisms described here are not only arguably the most popular methods in use today, but they are also the mechanisms at the core of both Datacomp and most if not all of the compression methods described later in the Related Work section.

There are many more compression utilities available than fundamental approaches to compression. MaximumCompression.com lists 125 utilities, but only 14 general techniques [10]. Salomon primarily divides compression techniques into the following categories: Run Length Encoding [7], statistical methods (such as Shannon-Fano [11] and Huffman coding [12]), transforms (such as Move-to-front [7] or the Burrows-Wheeler Transform [13]), and dictionary methods [7]. General techniques are often extended in various ways. For example, the Lempel-Ziv family of dictionary compressors (discussed later in this section) is made up of many variations on the same theme.[7]

On paper, specific compression algorithms are made of different compositions of these low-level techniques, sometimes reusing techniques in different stages. For example, bzip2 is based primarily on the Burrows-Wheeler Transform with Huffman coding, but also uses Run-length encoding in two different stages in combination with Move-to-front coding [14].

[7] Lempel-Ziv variations typically are named starting with LZ, such as LZ77, LZ78, LZW, LZH, LZO, LZMA, and many more.

Software implementations of a particular algorithm typically take the form of a carefully designed and optimized reusable library for use in applications.

This is not a research project into compression algorithms and utilities. However, it is about designing systems to use compressors wisely and judiciously. In order to illustrate why different kinds of data and system conditions impact the throughput of real-world compressors, this section will briefly discuss the common fundamental techniques and utilities seen in Datacomp and other AC systems.[8]

2.3.1 Run-length Encoding

Run-length Encoding is a simple and intuitive compression technique that attempts to save space by replacing sequential repetitions of symbols (runs) with one copy of the symbol and a number indicating the number of repetitions in the original text [7]. For example, given the string:

```
mississippi
```

… we can see that the letters ‘s’ and ‘p’ are repeated. We could encode this as:

```
mis2is2ip2i
```

However, this obviously doesn’t save any space, because one symbol (‘s’) and one digit (‘2’) still require two bytes. Even worse, if it is possible that the digit ‘2’ will appear in future input, we need some way to indicate that “s2” indicates a run length and not literal text. To signify this, we can use an escape symbol that we know will not be in the input. If we use the caret (^) as our escape symbol, our new, safer RLE-compressed string reads:

```
mi^s2i^s2i^p2i
```

[8] Much of the material in this section has been adapted from Salomon [7] and other online sources.

Unfortunately, this is now three characters longer than the original! This could have saved space if "mississippi" had longer runs of the letter "s" – but for our given input this method is unhelpful. As it turns out, while vanilla RLE may be effective for certain kinds of data, it is not especially useful for unvarnished English. As Salomon notes, "In plain English there are . . . many 'doubles' but a 'triple' is rare." [7][9] However, if we extend our system to consider repetitions of trigrams (a sequence of three letters) as opposed to single-letter repetitions we can compress the repetition of "ssi" and have the following:

```
mi^ssi2ppi
```

A savings of one measly letter!

While RLE is not especially effective on English, repeated bytes are common in certain kinds of data, especially sparse files or files with contiguous regions of identical data. RLE is sometimes used on data that has been preprocessed with a transform that increases the number or length of sequential runs.

Historically, RLE was heavily used to compress fixed-palette images, where linear sequences of pixels with the same color were represented by run lengths of identical bytes. As mentioned, RLE could be extended with the ability to identify ngrams longer than three letters, or word ngrams (repeated phrases). However, the value of this strategy is limited because RLE only encodes *sequential repetitions.* Data streams with repetitions that are not contiguous (e.g., multiple non-sequential appearances of the same word) would not benefit.

[9] In fact, Oxford Dictionaries says that triple letters do not exist in "proper words": http://oxforddictionaries.com/us/words/are-there-any-english-words-containing-the-same-letter-three-times-in-a-row

2.3.2 Move To Front Coding

Move to front (MTF) coding is a method of encoding data by replacing symbols in the data's alphabet with the index of the symbol in an alphabet, rather than codes indicating repetitions as in RLE. Given the string "mississippi," we can see that there are four characters in the alphabet: "m", "i", "s" and "p". We can represent each symbol with two-bit codes for the numbers 0-3 (rather than using 8-bit bytes). If we order our binary digit symbols alphabetically, (i.e., "i=00, m=01, p=10, s=11"), and we know that each index is two bits, "mississippi" could be encoded as "10330330220" (but in binary). The average value of the two-bit encoded symbols is ~1.6. If we took the time to order our alphabet in order of decreasing symbol probability – moving to the front the most common symbols – our alphabet would instead be ordered "s=00, i=01, p=10, m=11" (arbitrarily breaking the tie between "s" and "i"). Encoding "mississippi" with this alphabet gives the sequence "31001001221" which has an average value of 1.0.

However, assuming we do not know the symbol probability of the input and do not wish to perform a pass to obtain it, we can adaptively reorder the alphabet *as symbols appear in the input*, in an attempt to encode symbols using smaller average values. This is roughly analogous to the way that the most recently read books in a stack migrate towards the top. In this way, while a symbol may at first have a large index into the alphabet, immediately after it is encoded it will have a low index value because it will be "close to the top." Frequently-appearing symbols will "float" to the top, while rarer symbols will "sink" to the bottom. For example, Table 8 demonstrates adaptive MTF, starting with the symbols in alphabetical order.

Step	String	Alphabet	Index/Code
0	m	imps	1
1	i	mips	1
2	s	imps	3
3	s	simp	0
4	i	simp	1
5	s	ismp	1
6	s	simp	0
7	i	simp	1
8	p	ismp	3
9	p	pism	0
10	i	pism	1

Table 8. Example of MTF encoding.

The sum of the sequence “11301101301” is 12, for an average value per symbol of 1.09. While this average is not as good as we achieved when we knew the actual symbol probabilities, it is much better than simply using the alphabetical order (~1.6) and we did not need to scan the data beforehand. Obviously, the actual performance of MTF depends on the characteristics of the data, the starting symbol order, and so on. However, compared to RLE, MTF can benefit from local similarity even if the sequences of identical symbols are not contiguous.

2.3.3 Huffman Coding

Why is it useful to encode symbols with smaller binary strings (as we just took pains to do using MTF), if all the symbols require the same number of bits? The intuitive explanation is that if the more common symbols are encoded using smaller binary values, then the overall data can be made to take up less space (use fewer bits). This is because data encoded in this way compresses more effectively under Huffman Coding or other methods [7] that can use variable-length codes.

David Huffman presented his “Method for the Construction of Minimum-Redundancy Codes” [12] in 1952, an algorithm which improves upon Shannon-Fano coding [11]. With Huffman’s method, each symbol in a string is represented by a *variable-length bit string* where the most common symbols have shorter bit strings than less common symbols, and where the bit strings have the "prefix property."

The length property ensures that the most common symbols are encoded with the fewest number of bits (because they are first). The prefix property ensures that no bit string is the prefix of any other valid bit string (that is, no shorter bit string is the start of any longer bit string) [7]. This means that encoded streams can be parsed unambiguously without annotation. In other words, the prefix property enables bit strings to be decoded without special separators or escape characters that would reduce efficiency. For example, if A=0, B=10, C=110, and D=111, we can easily decode the string "111101100010" into "DBCAAB" because, reading left-to-right, there is no other way the string could be decoded even though no bits are explicitly used to indicate symbol changes. Even though variable-length prefix codes may in some cases be longer than fixed-length codes, they eliminate the regular cost of escape symbols necessary in many fixed-length schemes.

Huffman coding is widely used and is a step in both DEFLATE (i.e. ZIP and gzip) and bzip2. Classic Huffman Coding is optimal under certain conditions [12] regarding knowledge and characteristics of the data, although other compressors can compress more effectively when these conditions are unknown or do not hold. However, research into Huffman coding (and the creation of prefix codes in general) has created adaptive techniques and other extensions that can improve performance in suboptimal circumstances.

2.3.4 Lempel-Ziv (Dictionary Methods)

Dictionary-based compression was inaugurated with the publication of Jacob Lempel and Abraham Ziv's 1977 and 1978 papers, so influential that they are now known simply as "LZ77" [15] and "LZ78" [16]. Dictionary coding is a higher-level compression mechanism in the sense that it operates by identifying repeated strings of symbols in the text, replacing them with pointers to a dictionary containing their uncompressed counterparts. A common analogy for dictionary-based compression is to imagine compressing a text by replacing words in the input with the entry number of the original word in a dictionary (or word list). If a word does not exist in the dictionary, it can be emitted as a literal or potentially entered into the dictionary for future use.

Dictionaries can be built into the compressor (which would optimize them for certain kinds of input), can be created using an analysis step, or created on the fly. Commonly, dictionary compressors maintain a sliding window dictionary of past history that is reasonably large (e.g., 32K). This allows matches to be made based on repetitions in recently seen data, while capping the amount of memory required for encoding. While the dictionary may match many *potential* substrings during the encoding process, only the most efficient encodings (that encode the whole input) need be emitted.

For example, when encoding “mississippi”, the encoder would recognize that the string “is” is repeated in the text and store this in the dictionary as a potential encoding, but in the next step it would recognize the longer (and therefore more valuable) repetition of “iss”. As it turns out, the substring “is” in “mississippi” only appears as a part of the longer “iss” match. Therefore, there would be no need to write an entry for “is” in the compressed output, which saves space.

Where RLE has an extremely short “reach” for compression – repetitions must be sequential – and MTF has a somewhat larger reach (by adaptively optimizing the alphabet for the data), dictionary-based methods have a much larger reach in their ability to recognize longer matches from a larger stretch of the input. More memory and time devoted to compression improves the likelihood of finding matches (and enables "strength levels" for LZ compressors). Of course, this comes at a cost during compression in addition to a space cost; the final output must include the dictionary in addition to the compressed output. This "opportunity cost" of dictionary methods can raise the minimum amount of data required for compression to be successful, because compression must overcome the base cost of the dictionary. Finally, although compression can use considerable amounts of CPU time and memory, decompression for LZ methods can usually be structured to use a fixed or small amount of resources regardless of the "strength" used during compression.

It is hard to overstate the impact of Lempel and Ziv's contributions. From their work many variations have sprung, including LZW [17] used in the GIF [18] image format (and for many years under patent [19]), DEFLATE [20] (which combines LZ methods with Huffman coding), LZO [21] (a high-speed compressor using LZ techniques), LZMA [22] (a compressor designed for maximum compression ratio using LZ techniques and range encoding [23]) and many other algorithms and utilities.

2.3.5 Burrows-Wheeler Transform

In 1994, a paper describing the Burrows-Wheeler Transform (BWT) was published by Michael Burrows and David Wheeler [13]. Invented while they worked at the DEC Systems Research Center, the BWT is a transform intended to reversibly improve the compressibility of data. It is called a "block sorting algorithm" because it operates on long strings (i.e., "blocks") of input as a whole, as opposed to other techniques described here that operate on strings sequentially (usually at the byte level). In general, as with many other compression mechanisms, larger inputs tend to result in more advantageous outputs, but this is not true for all inputs.

The BWT is not difficult to understand, but it takes some time and space to describe. Since its internals are not the focus of this work, I recommend the descriptions available in Salomon [7] and Wikipedia [24]. That said, it is easy to describe why the BWT is useful: it tries to makes data more compressible by reordering the symbols in a block of input in a way that is reversible and which is likely to concentrate repetitions in the input. Unlike MTF or RLE, the BWT does not perform any compression at all. As a result, other mechanisms are required for compression after the BWT is used. Perhaps the most well-known compressor using the BWT is bzip2, which uses a combination of the BWT, RLE, MTF and Huffman Coding [12].

While the BWT can significantly improve compressibility, it requires a more significant amount of resources than the previous low-level techniques described here. One reason is that the BWT performs a large number of sorts and rotations on the text that requires significantly more computation (i.e., more passes over the data) than more localized methods such as LZ. Furthermore, while larger block sizes (up to some point) tend to result in greater concentrations of repetitions and thus better compression, this requires more memory – both for compression and decompression. Because of this, it is possible to compress data with bzip2 in such a way that a machine with less RAM will not be able to decompress it [25]. Nevertheless, the BWT is a powerful and popular technique for improving compression, especially as implemented in bzip2.

2.4 Application Requirements

Yogi Berra said, “In theory, theory and practice are the same. In practice, they’re not.” In the constrained and idealized world of theory, the same combination of input, method and execution environment obtain the same results. But systems exist in the real world, where there are often additional constraints that change the suitability of a choice for a particular purpose. In particular, compression must be used as part of some kind of application, which typically imposes its own demands.

Three higher-level issues that can affect the efficiency of compression are the ramifications of sequential versus random access, timeliness and the notion of effective throughput, and the interaction between compression and block-structured data. While the performance variations described earlier in this section were largely due to method, input, and environmental combinations, the following issues are primarily due to interactions between compression methods and the constraints imposed by the application employing compression.

2.4.1 Sequential vs. Random Access

Compression performance can depend on the mechanics controlling the input. For example, "regular files" have some convenient properties: they can be read a little bit at a time in sequence, they can be read all at once and they can also be randomly accessed. On the other hand, stream data (such as data being transferred over a network) is not by nature randomly accessible. Instead, it arrives in a linear fashion – the last byte cannot be read until the entire message has been received. Intuitively, random access approaches have an advantage in compression over stream-based approaches because they can consider arbitrary portions of the entire file rather than only a small portion of it. In essence, a compressor with random access to data can "see the future" as opposed to having sequential access only.

However, most compressors treat data as a stream for efficiency reasons, compressing data in relatively small sequential chunks rather than analyzing the entire input. This allows them to limit the resources consumed and guarantees that they will not need to backtrack. Treating the input as a stream, even if it is not, also makes the algorithms more flexible; stream-oriented compressors can process files as streams by simply reading them sequentially.

However, streams cannot be treated as files without buffering the entire input, which can be cost prohibitive in terms of time and space.

While the stream approach is a practical necessity, it has ramifications on compression performance. As shown in Table 5 and Table 6, breaking the stream of input into discrete chunks or blocks can have negative effects on compression ratio. Additionally, the nature of a stream means that future data is unknown. While it could be useful to know what bytes are arriving next ("Is this sequence going to be repeated again, or is this the last time?"), that information is simply not available in the stream context, which in turn limits the strategies available to a system attempting to make better compression decisions in real time.

Finally, random versus sequential access is also important for decompression performance. If a compression scheme lacks support for random access in decompression it can severely limit the usability of the method for typical applications. Consider a 100M log file that has been compressed to 50M. The compressed file is being read by an application that only needs the last 10K of uncompressed data from the original file. The application will pay a mighty overhead if it must read and decompress the entire input simply to read the last 10K of uncompressed data. One simple way to mitigate this issue is to quantize the amount of data compressed at once (e.g., by only compressing 1M at a time) and adding a table to identify which compressed chunks hold various byte ranges, but of course this can result in a loss of compression ratio due to smaller inputs and more metadata.

2.4.2 Effective Throughput and Latency

Compression is sometimes used to shorten the duration of I/O operations by reducing the number of bits that must be transmitted between two systems or subsystems. For example, compression is often used to improve throughput in network applications. Communication channels have a bandwidth in bits per second, and messages have a size in bits. Thus, compression can improve effective throughput if by reducing the size of messages it can enable more (or longer) messages to be sent in the same (or less) time.

Thus, improving efficiency is not simply a matter of compressing well or being fast – it is a combination of those properties and of the application requirements. For example, consider a situation where input is entering a compressor at some rate *A*, which is faster than the compressor's output rate *B*. If the output rate *B* is slower than the consumer's consumption rate *C*, the compressor will actually be the bottleneck reducing the *apparent output rate* of the system.

However, whether compression improves efficiency in the end depends on how well the data was compressed, how long it takes the receiver to decompress it, and whether this expansion will make up for the reduction in apparent throughput imposed by compression in the first place. If the compression ratio is poor or it takes a long time to decompress, compression will be a waste. On the other hand, if the data is compressed significantly and takes very few resources to decompress it, it can greatly improve throughput. Thus, *effective throughput* depends on all parameters that can affect compression performance: algorithm choice, data characteristics, base computational costs and environmental conditions, and the constraints imposed by the application itself.

The latency imposed by the compression choice is also an important component of the overall cost. If a stream *S* being transmitted between two hosts is compressed starting at time 0, the compressed stream *s* will not be available until some time after 0, because it must pass through extra computational steps. In other words, since compression requires computation, it will *delay* the data some amount of time, even if it ultimately improves throughput in the long run. This may or may not be a problem, depending on the application.

This delay can be thought of as one part of the compression time cost, which is composed of at least three parts: the base opportunity cost (the time required to compress one byte with any algorithm), the base cost for the particular algorithm being used, and the length-dependent cost for the algorithm and input in question. These costs reduce effective throughput and can eliminate the value of compression, even if the method being used "saves space."

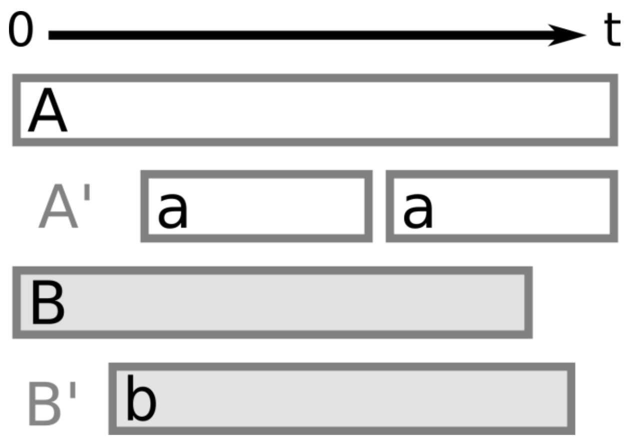


Figure 1. Compression ratio and compression time impact the effective throughput of transmission.

Figure 1 shows examples of these issues. *A* and *B* are two messages being sent between two systems using a communication channel with identical properties. *A* is sent at time *0*, and finishes at time *t*. The message *B* is shorter than *A*, so if it is sent starting at time *0*, it finishes earlier, at time *(t - n)*. Compressing *A* and *B* result in *a* and *b*, respectively. In fact, *a* is so small, it can be sent twice before time *t*, doubling effective throughput even though compression time *A'* delays transmission significantly. On the other hand, while the compression time *B'* is shorter than *A'*, message *B* does not compress very well. As a result, the transmission of *b* finishes *after* time *(t - n)*. In other words, although *b* is smaller than *B*, sending *B* uncompressed achieves a higher effective throughput. These issues apply to any scenario where compression is used in an attempt to improve effective throughput, because compression and decompression always impose some delay and the outcomes are based on a combination of factors.

2.4.3 Block Structures and Slack Space

It is often advantageous for computers to use block organizations for subsystems such as virtual memory, hard disks, and file systems. Hard disks have a minimum read-write unit (called a sector), which is usually 512 bytes, and file systems typically use 1K-8K organizational units called blocks (which themselves consist of two to sixteen physical disk sectors). Virtual memory systems manage RAM in pages, which are typically 4K or 8K. In addition to architectural reasons, block organizations have a number of advantages, such as reducing fragmentation and addressing requirements since every byte can be addressed relative to one of a much smaller number of fixed-size blocks.

However, block organizations create obstacles to the effective use of compression, because they can attenuate gains and magnify losses. When a block-based organization is used, compression only saves space if the data in *n blocks*, when compressed, can be written into *fewer than n blocks*. This can result in compression being a complete waste of resources, even if the compressed output requires fewer bytes. Even worse, if the output expands (for example, if a 1K block of input expands to 1.2K when compressed), it will now require two 1K disk blocks, wasting 800 bytes of so called *slack space*.

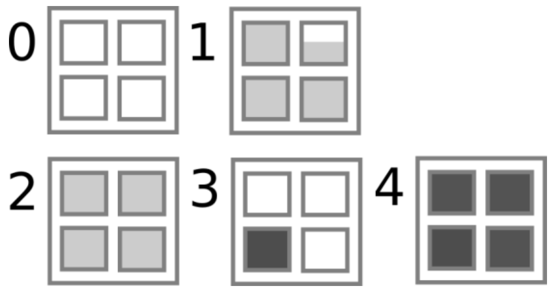


Figure 2. Each numbered square represents a 2K file system block. Each block is composed of four physical 512-byte disk sectors. Shaded portions of sectors contain data, while white portions do not contain useful information. "Slack space" occurs when the space occurs when the space in a block is not fully used, as in blocks 1 and 3.

Figure 2 shows five 2K disk blocks, which are each composed of four 512-byte disk sectors. Block 0 is empty, but four physical sectors are still dedicated to it. Block 1 holds a file that has approximately 1.75K of data, however the file system still requires a full 2K block, creating 256 bytes of slack space. Suppose block 3 contains a compressed version of block 1. Even though block 3 only contains 512 bytes of data, it still requires a full block, leaving 1.5K of slack space and saving no usable space on disk. As discussed elsewhere, compressed data can sometimes be larger than the raw data, and thus require more space. The cost of this result is amplified by a block-based architecture, because an increase in size of even one byte can end up requiring an additional block (for example, if compressing block 2 resulted in blocks 3 and 4). Thus, in a block structured environment, space is only saved when compressed data requires fewer blocks overall, such as if compressing blocks 1 and 2 together resulted in block 4.

Compression in block-structured systems is thus sensitive to the block size. As described in Section 2.2.4, larger input sizes tend to compress more effectively, so larger blocks should improve compression ratio on a per-block basis. But they also waste more slack space when blocks are not fully utilized (such as when a small file requires an entire block to itself). On the other hand, small block sizes reduce slack space, but they do not individually compress as effectively. It is often possible to mitigate these issues by concatenating several blocks and compressing them together in hopes that the resulting data requires fewer blocks for storage. However, doing this complicates data organization because it requires a mechanism to locate the files within the block, and may increase the cost to access a subset of any one of the blocks.

3 Adaptive Compression

3.1 Overview

3.1.1 What is Adaptive Compression?

“Adaptive Compression” is a term describing systems that attempt to use compression to dynamically maximize some property, such as throughput, energy efficiency or free space, by choosing the best compression method for the given input and environmental conditions. In contrast, most modern systems exist in what I call the “Static Compression” paradigm. In Static Compression systems, compression is not the default and is not flexible. Instead, compression is only used if a human being believes that a specific mechanism will be useful for a particular task over the long term. In this case, a developer might build compression into an application where it is always used, or they may make it an option that can be user-enabled on a case-by-case basis.

Another way of conceptualizing Adaptive Compression (AC) is that under AC, compression is not an exception, but the rule. AC systems *always compress*, attempting to use the most profitable method, which can change from operation to operation. Of course, sometimes no algorithm will be able to improve efficiency. But in continuing with the “always compress” philosophy, AC treats “no compression” as the fastest – and least effective – compression method.

Within this infrastructure, the challenge for AC systems is to make effective compression choices in an efficient fashion. Because making strategy decisions takes time, AC systems only improve efficiency if they save more resources than they consume. This challenge is complicated by the factors affecting compression performance (discussed in depth in Section 2) and those described in the rest of this section.

3.1.2 The Status Quo of Compression

In modern systems, compression is typically used in an "all or nothing" fashion, with human intervention often required to enable it, due to the potential cost of using compression at inopportune times. This conservative approach results in missed opportunities for applications to improve efficiency with compression. It also results in applications decreasing efficiency when compression is inappropriately enabled.

If compression is enabled (either by a person or an application designer), it is usually applied to all data for the duration of the application, locking the system into the compression choice regardless of changes in its utility over time. For example, OpenSSH does not enable compression by default. Instead, a user must request compression on the command line when creating a new session or edit a configuration file manually to change the defaults. Once enabled, compression cannot be disabled until the session is terminated. This is unfortunate since the conditions affecting whether compression provides a net gain or loss for efficiency can change due to dynamic conditions in the real world.

Some applications, such as OpenOffice, use compression as part of a file format, where all output written to disk is automatically compressed, regardless of its compressibility. A few applications, such as the Apache web server [26], allow experts to configure the system to apply compression more discriminately (by naming the file extensions to be compressed or excluded from compression). This is an improvement over blanket compression and can be close to the ideal in predictable environments, but the general approach is not responsive to changing conditions or to circumstances when the assumptions used (such as the availability of file extensions) do not hold.

Furthermore, if an application uses compression, there is a good chance that it uses DEFLATE, the same algorithm used in gzip, ZIP, and the HTTP specification. DEFLATE is a strong compressor with good throughput.[10] However, it is not always the ideal choice. Depending on the circumstances, a different algorithm could be more effective. This could be a fast algorithm that achieves less compression (such as LZO), or a slower algorithm that achieves better compression (such as bzip2). Simply using DEFLATE because it provides *some* benefit does not mean that it is providing the maximum benefit.

The main reason that the current approach to compression is static versus dynamic is that making the best choice is not easy. There are many potential compression methods to choose from, with many interacting factors that affect their performance, as discussed in Section 2. In many cases, a choice that would greatly improve efficiency in one circumstance would decimate it in another. Thus, the high value of performance that drives the use of compression use also keeps compression from being more fully utilized due to the risk of making a poor choice. This is probably one reason for DEFLATE's position as the *de facto* standard for compression – it is "average" in a positive sense. While other compressors may be faster, and still others may compress more effectively, DEFLATE strikes a practical balance between the two.

3.1.3 The Value of Adaptive Compression

As opposed to static strategies, AC saves time (and thus energy) by automatically using compression to reduce the size of communication payloads in a timely fashion when it would improve throughput, and by suspending compression when it would be wasteful. These kinds of savings can benefit not only typical network transmission, but many kinds of IO operations, such

[10] If you had to pick just one compression algorithm to take with you to a deserted island, DEFLATE would be a good choice.

as reading from and writing to disk. AC can also maximize the utility of limited storage devices by finding the ideal compression strategies for a given input. Finally, AC can save money in a direct way by making communication more efficient in environments where communication is economically costly (such as mobile devices). Unlike many hardware-based strategies for efficiency, Adaptive Compression requires only changes to software, so many existing systems could benefit from it immediately. Finally, because AC is a relatively unexplored optimization technique, its novelty increases its value as a research topic.

3.2 Compression Application Areas

While most people think of lossless compression simply as the main technique for improving the spatial efficiency of data, it is already widely used to improve throughput in several well-known areas – and a few surprising ones. If compression can be used to improve throughput in a given application area, then it is conceivable that *adaptive* compression could provide further benefits in these areas. To illustrate the wide array of potential applications for AC, I discuss several potential application areas in this section.

3.2.1 Compressed Communication and Network Transmission

While exorbitant per-byte cellular data costs provide a compelling economic motivation for network compression, non-adaptive compression for networking is usually done to improve performance in terms of time by reducing the number of bits that must be sent in order to transmit a message. Compression for networking is common, but is typically done in an all-or-nothing fashion. Devices, such as cellular phones, analog modems and fax machines include signal compression using low-level mechanisms like RLE and Huffman coding [7]. Applications such as OpenSSH [27] and Apache [26] can provide compression using the

DEFLATE [20] algorithm. In 2001, the Network Working Group of the IETF published RFC 3173 [28] describing IPComp, a standard for compression at the Internet Protocol (IP) level that includes regulations on handling compression in the face of fragmentation and so on.

Some implementations of basic AC systems are already in use. OpenVPN, an open-source VPN system, provides optional compression using LZO, which can be automatically enabled and disabled using a simple adaptive compression facility based on sampling [29]. Apache and IIS support DEFLATE and can be configured to eschew compression for certain file types [30] [26] (a rudimentary form of AC). IIS is able to cache compressed versions of documents, reducing server load for subsequent requests [30]. Finally, several vendors employ Performance Enhancing Proxy (PEP) [4] [31] [6] techniques, where compression is performed by vendors "in the cloud" on behalf of end users. Issues with these AC systems have already been mentioned in Section 1.

3.2.2 Compressed File systems

Compression is perhaps most commonly thought of as a tool for reducing the size of individual files. However, there are also many file systems that include compression in order to save space at the hardware level. There are two main kinds of compressed file systems: read-only, and read-write. In both types, the driver handles decompression transparently; applications use normal `read()` or `write()` system calls.

Read-only compressed file systems (such as SquashFS [32]) are created by using heavy-duty compression algorithms to maximally compress the input into a read-only file system. Read-only compressed file systems can appear writeable (such as for "Live CDs") by using a memory-resident overlay file system, although the changes cannot normally be made permanent. The benefit of read-only compressed file systems is that, because the data was compressed offline and without time constraints, expensive algorithms can be used to compress the image, maximizing efficiency.

Read-write, on-the-fly compressed systems are more relevant to this work. These systems appear to operate exactly like traditional read-write file systems, but compress and decompress everything transferred between the computer and storage device in real time. The low latency of these systems enable them to be used as primary file systems. However, in an attempt to avoid performance degradation, they may employ faster algorithms, which can result in suboptimal compression ratios.

A number of on-the-fly compressed (OTFC) file systems have appeared in the literature or on the market. For example, 1990 saw the release of both DiskDoubler and Stacker. 1993, Microsoft included a tool called DoubleSpace (later DriveSpace) in MS-DOS 6.0 that performed OTFC for FAT file systems [33] [34]. Most OTFC file systems are typically implemented as extensions to pre-existing file systems. Examples of modern file systems providing OTFC include e2compr [35] (an unofficial extension to Linux's Second Extended File System), Sun Microsystem's ZFS [36] (which also can perform the related but distinct service of de-duplication), and btrfs [37] [38] (supporting both the zlib and LZO algorithms).

OTFC file systems almost always improve spatial efficiency, which was much appreciated in the era of slow, low-capacity hard drives. Users were willing to pay a performance penalty because the cost of upgrading a hard drive was prohibitive. However, users sometimes found that OFTC *improved* performance[11]. This was because, like with compression for networks, OTFC could improve the effective read-write throughput for disk operations by

[11] I recall this from personal experience in the era when these kinds of systems were prevalent. However, it is hard to find references to anything empirical confirming this. Wikipedia also mentions this effect (https://en.wikipedia.org/wiki/Disk_compression#Performance_Impacts) but does not provide references.

encoding data more efficiently. This allowed individual operations to complete more quickly, because reading and decompressing a smaller number of blocks was faster than reading an equivalent amount of raw data (and vice versa).

In general, the same kinds of local computational conditions that affect compression throughput for networks, such as algorithm choice, data characteristics, input rate, resource contention, and so on, affect computational throughput for OTFC disk operations. Additionally, the block organization issues discussed in Section 2.4.3 and shown in Figure 2 apply to compressed file systems for performance as well. This means that many of the same techniques that can be successfully applied for networks can be used with compressed disks.

Unfortunately, to the best of my knowledge, all modern compressed file systems also perform compression in an "all or nothing" fashion without any adaptation. Instead, the file system "accepts" occasional degradation by relying on the assumption that compression will be advantageous *in the long run* and that if administrators are concerned about performance, they will choose faster algorithms or disable compression. This can be a lost opportunity for improved performance across a variety of metrics.

3.2.3 Compressed Memory

Through the use of OTFC techniques on system memory, computers can run software that otherwise would overflow the capacity of the physical memory. For example, if a computer has r megabytes of RAM, but program P requires R megabytes to run (where $R > r$), it may be able to compress P's memory (on the fly) down to r megabytes, allowing it to run (albeit at a performance penalty). In the 1990s, as computers became more powerful but RAM was still expensive, products such as Connectix's RAM Doubler provided this capability. However, now that RAM prices have dropped significantly, compressing RAM for space is primarily used for specific resource-constrained situations, such as embedded computing [39].

Obviously, RAM compression does not improve the performance of operations on the compressed data. However, it can be used to improve overall system performance by making more RAM available to other processes and improving the efficiency of virtual memory [40]. In these schemes, compression improves the *effective* throughput between memory and disk, improving overall performance. A number of researchers have successfully engineered compression capabilities into memory, including the processor cache [41] [42] [43] [44]. These systems can achieve meaningful gains through the judicious use of compression. However, they are also generally non-adaptive in terms of their compression strategy, which can result in suboptimal performance.

3.3 General Adaptive Compression

In the simplest sense, Adaptive Compression is an approach that makes dynamic compression decisions in order to achieve efficiency gains that would not be possible if compression were controlled by hand or with a static strategy. Because this is a complicated and unfamiliar problem with many components, it is helpful to begin with some definitions and terminology.

3.3.1 The Adaptive Compression Process

Imagine that two entities, a sender and a receiver, are in the middle of using Adaptive Compression to communicate a stream of data *D* over a channel *C*. The stream *D* can be divided into subcomponents $\{d_0 \ldots d_n\}$. The sender is about to send the next portion of data d_i, using the resources available to the sender and affected by any relevant system conditions, which we call its execution environment *E*. We can think of *E* as being composed of a set of resources and properties $\{e_0 \ldots e_n\}$, one of which includes the aforementioned channel *C* (which we could further break into its properties). The sender would like to maximize efficiency with respect to a resource such as time, space, or energy by using the "best" compression algorithm a_x from the set of available methods $A=\{a_0 \ldots a_n\}$ (where a_0 is "null compression," or sending the data uncompressed).[12] To make this decision, the sender must determine or estimate the system-level effect of executing each candidate algorithm a_i on the data d_i in the environment *E*, and make the choice expected to maximize efficiency for the selected resource.

Next, the sender performs the chosen compression operation, annotates the compressed data to indicate the method used, and transmits the compressed result to the receiver. From the sender's perspective, AC has been successful if the entire operation (selection, compression, annotation, transmission) was more efficient in terms of the critical resource than using a static, predetermined strategy, such as not compressing or using DEFLATE. For this to be true, the adaptation costs (choice selection and annotation) and the compression costs (using a_x in environment *E* on data d_i) must have resulted in a net savings of the critical resource.

The receiver then reads data from the network, decodes the annotations to determine the compression method used, and decompresses the data. Even if the receiver did not participate in the decision process, the receiver would typically benefit from AC if it is concerned about the same resources as the sender. When optimizing for space, both sides see the same benefit, and since decompression is typically faster than compression, if a strategy saves time for the sender it will probably save time for the receiver (if the computational resources for both systems are

[12] This work focuses almost exclusively on optimizing from the sender's perspective. This description of AC could easily be expanded to include cooperation in strategy selection between the sender and receiver, but that topic is beyond the scope of this project.

similar). That said, the receiving side of an AC communication has much less control in general, and is typically limited to deciding how to perform decompression (in serial or parallel, in order or out of order, and so on).

3.3.2 Elements of Adaptation

Due to a lack of terminology, I chose a set of terms to describe specific elements of the AC problem space in hopes that this might facilitate discussion and communication. These terms, which are summarized in Table 9, include: Methods, Data, Environment, Opportunities, Events, Results, Scenarios and Strategies. I define and explain each of these terms below.

3.3.2.1 Data

I use the term Data to mean the input being communicated over an adaptively compressed channel and which will be compressed by the AC system with some compressor. In this context, Data refers specifically to the discrete chunk of input being discussed or considered for compression, which may be a small subset of a larger input. Different Data are distinguished by having unique properties which may affect compression outcomes, such as differing information content or lengths. It is conceivable that two distinct pieces of Data might have different information content but identical compression outcomes for all compressors (e.g., a transposition of two bytes). However, since it is unclear how different two inputs can be while remaining the same from a compressibility perspective, we generally treat any identifiable distinctions between Data as potentially significant.

3.3.2.2 Methods

Each distinct compression mechanism is called a "Method." A single Method is defined as one unique choice drawn from the set of all options and parameters supported by the AC system. For example, one example of a Method is “gzip, strength level one, input size of 256K, two threads.” A different method is “gzip, strength level one, input size of 256K, **three** threads.” These are different Methods because their performance characteristics (e.g., run time or compression ratio) could be distinct even though the compression algorithm itself is the same. Similar to Data, if “two threads” and “three threads” *always* have identical effects in every circumstance we would prefer to treat them as the same abstract Method. However, since I know of no practical means to evaluate such a similarity, I treat as distinct any combination of parameters that might reasonably be expected to provide different outcomes.

3.3.2.3 Environments

Similar to how discrete types of Data and Methods are distinguished by their properties which (may) affect compression outcomes, we use the term Environment to define the set of all execution environment characteristics (the system hardware and software state) that could affect empirical compression outcomes (how quickly the data compresses) – or the practical value of those outcomes (how much energy is saved through compression). These characteristics can be further divided into *fixed characteristics*, such as the number of processing cores the system has or the physical channel’s bandwidth, and *transient characteristics*, such as the current CPU utilization or available bandwidth.

3.3.2.4 Opportunities

I use the term “Opportunity” to describe a juncture at which the application of a compression Method can occur. An Opportunity is defined as a relationship between some Data (the input) and the Environment in which compression and communication will take place. In the same way that an input could be cut into different overlapping Data elements (such as two

128K or four 64K pieces), the set of *possible* Opportunities given some Data and Environment may overlap.

3.3.2.5 Events and Results

A compression "Event" is the unique application of a specific Method for a particular Opportunity. In other words, an Event is the use of a Method on some Data in a specific Environment. Each Event produces a set of "Results," which are the meaningful outcomes of a particular Event, such as the run time, achieved compression ratio, and so on. Some of the individual Results depend only on the Method and Data (such as the compression ratio), while others depend on the Method, Data, and Environment (such as the run time). It is not apparent whether there are any meaningful Results which depend only on Method and Environment.

While we can informally think of the compressed or decompressed output as a kind of "result" of the event, it is not formally one of the Results. This is because in my conceptualization, Results are not formally tied to the semantics or information content of the Data, but to the statistics that are relevant from an adaptation perspective. If the compressed output was one of the Results, then every unique file would have a unique result even if all other measurable statistics were equal. By separating the Result from the content of the data, Events that produce the same Results can be identified as somehow being similar. This similarity can be useful in terms of making compression decisions.

At any rate, the Results of an Event will determine whether a compression operation was worthwhile based on the priorities and goals of the user or application. Additionally, the creation of an Event (by executing a Method on an Opportunity) may exclude other potential Events by virtue of consuming some of the Data used in other potential Opportunities. (These Opportunities will never be used, since each byte of Data is typically only compressed once.)

3.3.2.6 Scenarios and Strategies

A "Scenario" is a higher-level, longer-term task, such as copying a particular file over a network under a set of environmental conditions. In its simplest form, a Scenario is a sequence of Opportunities. However since the application of one Method may change the set of possible future Opportunities, we can also think of a Scenario as being the "problem space" of the AC optimization problem.

This space is composed of all possible sequences of one or more related, but discrete Opportunities and their underlying Data and Environmental properties. In this sense, the aggregate properties of the Opportunities (to the extent that they are known) determine the properties of the scenario. For example, "transmitting encrypted data over a network" describes a scenario wherein at least one characteristic of all the Opportunities is defined – the Data has been encrypted and is thus unlikely to be compressible.

In the same way that the Scenario defines a set of Opportunities, a Strategy describes a set or sequence of Method choices. A Strategy is typically oriented towards a particular goal in a particular Scenario, but does not necessarily need to be determined down to the specific Methods used or how they are selected. For example, we can talk about finding the best Strategy for the Scenario "transmitting Facebook data over a network at 1Mbit/s while maximizing for energy."

Table of Adaptation Elements

Adaptation Element	Description
Data	The properties defining a finite portion of input to be transmitted via an AC system.
Environment $= \{p_1, p_2, \dots p_n\}$	The set of environmental properties relevant to compression performance (e.g., number of cores, CPU load, available bandwidth).
Opportunity = (Data + Environment)	The juncture for compression created by the combination of any input and environmental parameters that affect compression outcomes.
Method	A specific, unique compression operation. A member of the set of all available Methods in a system.
Event = Method(Opportunity)	The application of a specific compression Method on an Opportunity
Event $\rightarrow$ *Result_Set* $= \{r_1, r_2, \dots r_n\}$	Results are products of Events, including statistics such as the compressed size (compression ratio), compression time, CPU utilization, etc.
Scenario $= [\mathit{Opportunity}_1, \mathit{Opportunity}_2, \dots \mathit{Opportunity}_n]$	A sequence of Opportunities arising from a long-running communication, such as downloading a web page over a slow link.
Strategy $= [\mathit{Event}_a, \mathit{Event}_b, \dots \mathit{Event}_z]$	A collection of Methods meant to optimize the efficiency of a Scenario for a particular resource.

Table 9. The elements of Adaptive Compression choices.

3.3.3 Components of an Adaptive Compression System

I also developed a set of terminology for the AC solution space. Every AC system that makes decisions based on information in some operational context has analogues for at least four high-level components: Methods, Monitors, Models, and Mechanisms. These components (or some entity taking their place) are necessary for adaptation. They also define the major sources of cost for an AC system.

Methods are, as described in Section 3.3.2, the set of discrete compression tools available to the system. Monitors are the software components responsible for determining the properties of Opportunities as well as acquiring the relevant Results following Events. Models are the decision-making components of an AC system, and Mechanisms are the means by which AC interfaces with the communication system.

3.3.3.1 Methods

In order to be adaptive, an AC system must have the ability to apply a method or methods in a non-static fashion with the goal of optimizing for some particular purpose. In most AC systems this is done by supporting at least one actual compression method and one "null compression" method. However, as described in Section 4.2.2, some AC systems only apply one Method but adaptively determine *when* to apply it.

A simple AC system could choose between null compression and DEFLATE, such as Apache configured to compress all data except audio or video files. However, since different Methods have varying Results given some Opportunity, AC systems often include a broader selection of compression algorithms meant to provide a range of Results. This can be performed by varying the strength level of a single compressor, or by using compressors with different properties including "heavy duty" algorithms such as bzip2 and xz (lzma) or algorithms designed for speed, such as LZO. Compared against DEFLATE, the former two algorithms offer improved compression at a cost of time and energy, while the latter algorithm is extremely fast but a poor compressor. While there is no limit to the number of options a compression algorithm might use, in this work we focus on basic choices such as the underlying algorithm (e.g., DEFLATE vs. LZMA), the length of the Data selected for compression (the "chunk size"), the "strength level" (if the Method in question supports it), and the number of compression threads used.

3.3.3.2 Monitors

AC systems must be able to identify the relevant Data and Environmental properties of the Opportunities encountered in order to make informed compression decisions or to tune future decisions. I use the term Monitors to describe the system components used for this purpose.

Monitors can be “one-liners” that simply report event Results, functions that obtain some value, or a daemon-like component (internal or external to the application or system) that might literally monitor some condition. Monitors can report empirical values or predict values based on observations. Examples of Environment Monitors include modules to track CPU load, frequency scaling for systems supporting Dynamic Voltage/Frequency Scaling (DVFS), and the throughput of the communication channel.

Monitors must be efficient, since they (along with Models and Mechanisms) are the “adaptivity costs” that an AC system must pay, even if null compression is used. While these costs can be minimized in various ways, they must continue to be paid in order to be ready for changes in conditions. For example, it is critical to monitor or estimate the compressibility of input. If data is recognized as uncompressible, then an AC system can “fall back” to null compression without spending any additional resources. However, some monitoring costs must still be paid, in case the data becomes compressible in the future.

In addition to the accuracy of individual outputs, the frequency (i.e., “sample rate”) and resolution of a Monitor is important. For example, if a Monitor can estimate the compressibility of input for every 32K, it can lead to more finely-grained strategy choices than if it can only estimate for every one megabyte of data. Of course, greater resolution and accuracy typically comes at the cost of more execution time.

3.3.3.3 Models

Models are the “brains” of the AC machinery. Monitors comprise the input to Models, while Models choose the Method – the strategy choice to use for a given Opportunity – based on that input. Models can be extremely simple. A simple and direct model might choose the strength level of a compressor based on a single environmental property, such as the current CPU load. Models can also be complex, such as a set of equations or a database of some kind. Models can be dynamic, where the results from a particular Method and Opportunity are integrated into the model for future operations, or static, where the Method selected for a given Opportunity never changes.[13] The costs of an AC system's Model (or Models) are also important to minimize because, like Monitors, they may use resources even when the system is not able to improve efficiency.

3.3.3.4 Mechanisms

The last component of AC systems is the Mechanism they employ. The Mechanism consists of the software machinery necessary to make AC functional and implements whatever interface exists for interacting with the system. Practically speaking, this consists of a large amount of mundane code. From an academic perspective, we are most interested in the way that AC is integrated into the system. For example, existing AC projects function at many system levels; some use user-space libraries, Grid Computing middleware frameworks, Java virtual machines, kernel integration, third party proxies, and even a distributed AC system built into a network. In some systems, the Mechanism does not have a significant impact in how the system functions or is designed. In other cases the Mechanism has a meaningful impact on the kinds of choices the system can make, what its costs are, how users interact with it, and so on.

One part of the Mechanism that most AC systems require is a messaging or protocol format. In non-adaptive systems (or systems that always use one compression method) assumptions can be made about the contents of a communication, and no special indication must be made to communicate this information. In contrast, many AC systems modulate their use of

[13] Apache's use of DEFLATE is one such model; compression can be conditionally selected (and thus adaptive in a weak sense), but the conditions are never altered.

compression in a way that requires annotating their output with decompression instructions for the receiver. As a simple example, in the HTTP specification [45], the *Content-encoding* header indicates whether data has been compressed with DEFLATE. If the header indicates compression, a decompressor will be used on the client side.

Like Monitors and Models, the messaging component of the Mechanism adds overhead to AC systems even during periods where strategies do not change, because the system must be prepared for the *possibility* of a strategy change. However, the messaging overhead is typically quite small. For example, consider one megabyte of data (1,048,576 bytes) cut into sixteen 64K chunks. If the Message overhead per output chunk is 32 bytes, then the total overhead is only 512 bytes or approximately 0.05% of the total.

More significantly, the abstraction layer in which an AC Mechanism resides can have a meaningful impact on cost, complexity, accuracy and portability. Designing low-level kernel code is expensive from a human perspective, but offers the best performance, is transparent to the application and has unrestrained access to key system data. Developing at the user level is simpler, but the separation from system internals means that the system will require code changes in order to be used and will need to cope with partial knowledge in addition to system call overhead. Middleware may allow a compromise between these two options if it exists. It can make the mechanism transparent and might also provide additional facilities useful for adaptation (e.g., parallelism), but these add dependencies to the system and may make it less portable to other contexts. We will discuss these issues in detail in the Related Work and Datacomp Design sections.

3.3.4 Choosing a Compression Strategy

In theory, every discrete Event (operation of a Method on an Opportunity) gives rise to the same Result set. This is because each Event is defined by every meaningful property of the Data, execution Environment, and Method. If two apparently identical Events obtain different results, then they were by definition not the same type of Event. This means that, in theory, if we know the Results for a particular Event through analysis or past experience, we therefore know the Results for *future* instances of that event. If we knew the Results for every type of Event we could simply choose the best Method for every Opportunity.

But, as usual, the devil is in the details. There is no comprehensive Model naming the best choice for every possible Opportunity, and it is unlikely that such a database could be constructed. Even if such a thing could be created, it would likely be prohibitively large due to the vast number of possible combinations of properties. This means that any Model we have or construct is almost certainly going to be approximate in many ways. Thus, a major challenge is developing a Model that maximizes accuracy while minimizing cost to use and maintain.

Another obstacle is that at any given Opportunity, some of the most meaningful properties of the Data and Environment may be unknown at the time a choice must be made. For example, while properties of the environment such as the CPU load can be read almost directly by Monitors from system facilities, predicting the "compressibility" resulting from some Event requires analysis of the data. Unfortunately, techniques for performing this kind of analysis efficiently and accurately are not well-explored.

Other challenges are more esoteric and no less difficult. Theoretically, the result for a particular Event (a combination of an Opportunity and Method) is always the same. For example, if Method *X* is *always* the best choice for Opportunities of type *Y* (as "null compression" is for Opportunities containing encrypted data), then the ideal strategy for *Y*-Opportunities is known. However, we do not know the actual set of properties essential for defining unique Opportunities, which means that we cannot be certain, if presented with two Opportunities of apparent type *Y*, whether they truly are the same type (and thus would obtain the same Result given the same Method) or whether there is some unknown characteristic that distinguishes them. In combination with the Model complexity problem described above, this means that our models must cope not just with approximation, but also with error, both of which reduce the potential benefit of AC.

Finally, when thinking about making AC choices, it is important to define the scope of a particular Scenario. If a Scenario is extremely short or well-known, such as the transfer of a specific file type across a network, the likelihood that a known best static solution exists is increased. However, when we consider real-world scenarios, such as traffic on a VPN used by many users, the diversity and variability of the Scenario's properties quickly becomes too great, unpredictable, or costly to analyze in this deterministic way. Even if the solutions for certain kinds of Scenarios are known in advance (such as encrypted data), in practice there are many cases when the nature of the data in an Opportunity is unknown (or may be known at some points but not others).

3.4 Requirements for Profitable Adaptive Compression

Based on the information discussed in this section, we can identify two requirements for AC to be the best choice (as opposed to a static approach) for a particular Scenario:

1. A dynamic strategy is necessary to achieve optimal efficiency for the Scenario, and
2. There is a cost-effective adaptive technique suitable for the scenario.

We can also say that AC may be the best *practical* choice for a Scenario when the ideal solution is *not known* (and the second condition holds). I will expand on these requirements in the following sections.

3.4.1 Optimality Requires Adaptation

There is a meaningful difference between saying that a certain strategy is the ideal choice and saying that some strategy is the best known choice. In this manuscript, the term “static strategy” refers to the use of a single compression choice for an entire Scenario (e.g., “always use DEFLATE”). If a single static strategy is ideal for a given Scenario, then AC cannot be the ideal choice. This is because the added computational cost of an adaptive strategy – the costs of the Monitors, Models, and Messages – are simply unnecessary in such a Scenario even if minimized by clever design. Put another way, AC is never free. It can only be the ideal choice if a static choice is not.

In some circumstances, it is possible to determine an ideal static solution for a sufficiently constrained Scenario if enough relevant parameters of the constituent Opportunities are pre-defined (can be obtained without a Monitor) and can be assumed to be fixed. For example, a Scenario transmitting encrypted data over a network will have a single static ideal solution regardless of length, network, or execution characteristics, because the data is not compressible. But we need not be limited to worst-case Scenarios to make these claims. Given any reasonably homogenous type of data (like machine code, HTML, or English text) and a static environment, it is likely that there is one compressor that performs the best on this type of data, even if the documents being transmitted are not identical.[14] If the data is compressible, then performance should be improved (regardless of whether the system was adaptive).

On the other hand, if we know that a static solution is *not* the best, then we can say that AC is the *ideal* choice. Proving this, however, is not normally practical. More commonly, we simply do not know what the best strategy is. In these cases, we say that AC *may* be the best choice. AC is attractive precisely when we do not know of a good static solution, and we believe that an automated process can do better than the best guesses of users and developers.

3.4.2 Cost-effective Adaptive Compression

In addition to a Scenario that can be best served by a dynamic solution, the effective use of AC requires some kind of adaptive technique profitable for the Scenario. Because AC is a young field, it is not always known whether such a technique exists for a given Scenario. For example, consider a Scenario transferring data that is uncompressible 99% of the time, but 1% of the time is highly compressible. Unfortunately, the time when that 1% is transmitted is not known. Further suppose that compressing all the data with even the fastest compression method reduces throughput due to the pathological 99%, even though it captures the potential benefit of the 1%. If we do not know how to find that 1% in an efficient fashion, we cannot benefit from adaptation. We would be forced to send it uncompressed.

Even if there is a way to identify the best compression strategy, the Monitors and Methods used must be cost-effective enough in terms of the resource being optimized that they do not consume all the benefit that would be provided by compression. For example, one effective method for finding the compressible 1% of the data would be to compress every 1% of

[14] Since I classify manual selection with other static strategies, I do not consider a human being choosing the compressor ahead of time as being "adaptive."

the input until we find what we are looking for. While this would be extremely costly in terms of time and energy, it would be worthwhile if space reduction was of critical importance.

Finally, it bears noting that a "cost-effective adaptive compression technique" need not and in fact cannot obtain *all* of the possible efficiency improvement, because AC itself has a cost. Rather than being ideal, AC must simply be able to do better than any known static strategy while covering its own costs to be worthwhile and cost-effective. And from a practical perspective, it must be able to do *sufficiently better* than the alternative to be worth the added complexity.

4 Related Work

My focus is on a general, adaptive, and modular compression facility that is useful for improving efficiency in terms of time, space, and energy, and designed to be profitable in everyday, general purpose computing environments. While there has been a string of successful and interrelated projects created over the last 15 years or so (summarized in Table 10), Adaptive Compression has to date not been an especially well-developed field.

Existing systems employing AC tend to be highly tuned for specific constraints and application areas, which is understandable because this simplifies the required solution. However, as a result, solutions can heavily depend on the specific properties of the problem and available information given the abstraction layer the system is built into. This leads to uncertainty about the extent to which such a system would perform in a different environment. Furthermore, while each prior work has successfully demonstrated success within specific areas or sets of problem constraints, most of these areas are not well explored beyond those first seminal works, and there are many subcomponents of AC systems (such as compressibility prediction mechanisms) that are virtually unexplored.

Additionally, while every AC system I reviewed found success in improving efficiency, those successes were heavily dependent on the environment in question, the assumptions underlying the project, the experimental design and the test data used. In particular, the data used for most prior AC system evaluations is generally compressible and did not vary especially widely. This is perfectly acceptable for an experimental environment with relatively controlled data, however in such scenarios the task of adaptation is consequently easier because compression is often the right choice and the benefits are significant.

This is not meant as a criticism of prior systems. Rather, simplifications are inevitable in research because there is simply not enough time or resources to explore every dimension as fully as we would like (not to mention exploring as fully as others would like). However, because of the constraints and limitations of past work in this area, it is not clear whether the mechanisms and successes of past AC systems are directly transferrable to general computing systems of today. That said, past successes do demonstrate clearly that AC can "squeeze water from a stone" and improve efficiency in various environments. Given the explosion of computing in the real world using mobile, wireless, and personal computers and the importance of many technological and environmental energy issues, prior work suggests that AC could be a valuable tool for improving the efficiency of general computing devices.

4.1 Compression Studies

A number of studies have investigated compression performance over a range of inputs [46] [47] [10] and some have investigated the notion that static compression strategies are suboptimal [3] [46]. Other studies have considered the suitability of particular heuristics for driving compression solutions [48] [49] [50]. In the Fine-Grain Mixing papers [49] [50], researchers explored mechanisms for making compressor performance more energy and throughput proportional by emphasizing mixing compressed and uncompressed data, rather than adjusting compression "strength", in order to linearly scale up compression with computation cost. Michael Gray showed that this mixing can be done in a way that is backwards compatible with the existing zlib implementation [50].

While compression for downloads has long been recognized as valuable (it is included in the HTTP standard [51]), Wang and Manner showed that compression for uploads can be energy-beneficial, even for low-power smartphones [52], and found that both network conditions and data type should be considered in decision making. Kothiyal, et al. [46] demonstrated that compression can be useful to save energy on servers for file I/O, but that energy savings were dependent on many variables, including compression algorithm, data and workload characteristics, and hardware. The notion of using parallelism for compression to improve throughput is not new [53], although it has not been very well explored in the Adaptive Compression literature.

4.1.1 Communication vs. Computation Costs

Barr and Asanovic's influential study [3] into energy-aware lossless data compression is cited by almost all of the AC literature. Their early study presented a comprehensive exploration of the energy costs and benefits of compression. They carefully measured computational energy cost versus communication cost and showed that, on their platform, roughly 1,000 computations could be performed on a bit for the same cost as transmitting one bit across the network.

Ultimately, their paper demonstrated that carefully balanced decisions are necessary in order to make the most energy-efficient compression choices, and that the right choices are not always obvious. However, they did not develop an adaptive system, per se. Furthermore, while they performed highly detailed analysis of their experiments, they investigated fewer variables than I am, and used a platform (the Compaq iPaq [54]) which is not only old by today's standards, but differs significantly from my target platform of typical modern end user computing devices. This is relevant because performance characteristics are critical for AC system design and evaluation.

4.1.2 Compressibility Prediction

Finally, efficient prediction of compressibility is especially poorly explored. The core concept is that while the effect of compression can be accurately measured by actually performing the compression operation, this is too expensive to be widely useful for making compression decisions. After all, the whole point of making decisions adaptively is to make the best choices, including not compressing when it is wasteful. Instead, compressibility prediction involves using extremely fast algorithms that might not perform compression but output a value indicating how well the data is likely to compress.

Most AC systems rely on sample compressing (i.e., performing compression on a small portion of the input), recent compression performance, or monitoring general system throughput to indicate how well compression is working. William Culhane, in his senior Honors Thesis at Ohio State University [48], designed and performed the only "efficient compressibility prediction" experiments I have found. These experiments evaluate the predictive power of three analytic mechanisms versus the actual compression outcomes of three basic algorithms. One of his expressed purposes was using the predictors to choose the algorithm to be used for compression (although he did not appear to build an AC system using the predictors).

The three algorithms he tested are 12-bit LZW, Variable Length 12-bit LZW, and Huffman Coding. The three mechanisms used for prediction were the "standard deviation on the bytes, standard deviation of the difference of consecutive bytes, and standard deviation of the XORed value of consecutive bytes" [48]. Each of these analyzers was run on input that was then compressed, after which a model was created to relate to compression performance to the output of the predictors.

These three mechanisms were chosen as a measure of “the evenness of the distributions” of the bytes under the supposition that this would have an effect on compressibility. The first measure considers the probability of each byte in isolation, while the differencing predictors relate pairs of adjoining bytes in the hopes that this would incorporate some of the structure of the data into the prediction. Ultimately, the basic standard deviation was most successful for all three compressors, with his results demonstrating that prediction is possible (even predicting which algorithm to use) under certain circumstances.

While I developed my method independently, my compressibility prediction mechanism is similar to a simplified version of Culhane’s “standard deviation of the bytes” approach. As discussed in Section 6.4.4, my method counts the number of bytes that appear frequently relative to the length of the data, with the similar goal of determining how evenly distributed the input bytes are. The benefit of my approach is that it is computationally less expensive than computing the standard deviation, which is key for an AC system like Datacomp. Nevertheless, I believe William Culhane deserves recognition for his groundbreaking work in this area.

4.2 Existing Adaptive Approaches

There have been a number of implementations and prototypes of AC systems, which often incorporate existing algorithms and results from previous studies in addition to new research. A summary of the systems surveyed in this section appears in Table 10. These systems are similar in spirit to my work. However, these works have typically investigated adaptive compression for a specific purpose, such as optimizing network use for the transmission of large files in Grid Computing environments, network monitoring, satellite networking, and so on. These systems do not target the "wild west" of general end user computing. This does not mean that they are less effective, but because these systems primarily focus on specific application areas, they often rely on one or more elements or design decisions appropriate for the area, but which potentially limit their broader use or ability to maximize efficiency.

These limiting elements include (but are not limited to): subsystem-specific or hard-coded proxy measures for efficiency (which may not be ideal or are not transferable to other subsystems), dependence on exotic external services that are not available in the real world, pre-calculated decision tables or models (which may be specific to particular hardware or algorithms), reliance on particular compression algorithms due to assumptions which may not hold for other compressors, limitations on parameters such as block size or compression level (either for simplicity or because of dependent assumptions), and more.

Table 10. Summary of AC systems surveyed.

System	Year	Methods[15]	Mechanisms[16]	Monitors	Model	Notes
Knutsson and Bjorkman [55]	1999	Eleven modes from Zlib: 0 (none), two lightweight methods and the standard levels 1-9. Variable chunks.	Built into the TCP protocol of Linux 2.0.30. TCP option specifies compression capability.	Kernel-visible write_queue length.	Empirically tuned function choosing compression level based on queue length, with smoothing (lock-in).	
DCFS [56]	2001	JAR [57], PACK [58] and TGZ. (tar.gz) Also "none" in extension (see notes). Compresses whole .jar file.	Special Java class loader to download and decompress Java libraries.	Size, predicted decompression time and predicted network performance (via probes or NWS).	Estimates and minimizes "total delay" via network performance prediction and average decompression time.	Extension to compress on the fly also considers compression time on server and beats *uncompressed* Java libraries.
NCTCSys [59]	2001	None, gzip or bzip2 (with and without LIPT [60]). Whole file.	Application level with caching; Header invokes dictionary sharing protocol for LIPT	Bandwidth, Line speed, number of clients connected, server load, file size.	Threshold levels directly select compression†	Designed for text compression within applications (e.g., FTP, HTTP). Compresses *before* `send()`.
ACE [61] [62]	2003	None, gzip, bzip2 or LZO. 32KB chunks.	ORP Java JVM modifications to capture `connect()`, `accept()`, `read()`, `write()`, `send()`, and `recv()`. 4-byte header specifying compression method.	Current and predicted CPU load and ABW at both hosts (via NWS). Compressibility estimated via "last block" assumption (periodically forced if needed).	Choices based on precomputed linear models relating algorithms and data to monitored resources. Thresholds and cooldowns for edge cases.	No compression (NC) if input < 32KB, uncompressible, or low CPU. Precomputed and fixed linear relation model between compressors. Requires NWS.
Xu [63]	2003	None, gzip -9, Zlib -9, Compress -16, bzip2 -9. Only one used at a time. Chunk size is compressed data blocks.	Compression on proxy; decompression apparently within test application.	Known fixed bandwidth, file size and block-level compression gain (known due to precompression).	Threshold model chooses when to send compressed data (manually tuned for hardware and bandwidth).	Mechanism for saving RX energy via compression on proxy. NC if input < 3.8K bytes. Assumes known, static bandwidth.
Wiseman [64]	2005	None, Burrows-Wheeler Transform ("SGI version"), LZ with Huffman [65] and Huffman coding. 128K block size.	Integrated into IQ-Echo grid middleware, using "handlers" for de/compression and "attribute" feature for communication.	CR estimated via 4K samples. ABW, RX-rate and CPU monitors (partly via middleware).	Fixed hierarchy of algorithms with thresholds parameterized by current system state as related by monitors.	Approach extensible to other middleware. Block and sample size tuned empirically.
AdOC [66]	2002	None, LZF [67][17] and the gzip levels, representing levels (0-n) respectively. Fixed 80K minimum chunk size.	User library providing functions with POSIX-like API. Also integrated into grid middleware.	FIFO queue length and rate of change. Detects ABW for gigabit networks. Performs a type of CR sampling.	Increase compression level if queue growing; decrease if queue shrinking. Thresholds and cooldowns for edge cases and weights for rapid queue length change.	NC if input < 80K (or 512K if ABW > 500mbit/s). Unclear if levels used extend beyond gzip -6.

[15] Method colors: Blue: zlib strengths or zlib strengths and LZF (two related projects). Green: chooses from more than two Methods. Red: binary choice between "none" and one compressor. White: Uses only one compressor but dynamically retrains system.
[16] Mechanism colors: Yellow: kernel or virtual machine. Gray: Application layer. Orange: Proxy. Purple: Grid middleware. Tan: networking infrastructure.
[17] LibLZF is an extremely fast compressor with LZO-like speeds [65] but which is BSD- rather than GPL-licensed.

Maddah and Sharafeddine [68]	2008	None and "Zip Level 1" as defined by #ZipLib [69]. Investigated effect of 4K, 8K, 12K, and 16K chunks (but size at runtime is fixed).	Built into file-sharing application for mobile devices.	Wireless radio signal strength and known (fixed) bandwidth.	Compress (with a fixed chunk size) only when radio signal strength is below a precomputed threshold.	Evaluated Zip Level 9, bzip2, gzip, and Deflate from #ZipLib [69] for performance but only used Zip Level 1. No compression for known hard extensions (e.g., MP3, GIF, JPEG).
Politopoulos, et al. [31]	2008	Tested none, rar, rzip. LZO, bzip2 and gzip (the last three in both "fast" and "best" modes). Used LZO in ~64KB chunks.	Built into DiMAPI [70] network monitoring tool. Data buffered and compressed according to model.	Buffer space, packet inter-arrival rate and average throughput.	Buffer compressed with LZO and sent when full or if time limit (based on inter-arrival rate and throughput) has expired.	Very similar method used in [71] for VSAT communication with de/compression occurring at ground stations.
IPZip [72]	2008	Gzip. Chunk size based on adaptation duration (not specified).	ISP-level tool that creates a compression plan by dividing and reordering packets to optimize compression using gzip (in this work).	IP Header information. Recent compression ratios.	In online mode, worsening CR triggers retraining.	Reorders data after decompression. Offline mode searches for best classifier.
Shimamura [73]	2009	None and LZO "fast". Packet level compression.	ISP-level. Compression by "advanced relay nodes" at the packet level. Simulated in ns2.	Queue wait time, expected compression cost and ratio (simulated), packet size, output bandwidth.	Compresses if an equation model using monitored values is satisfied.	Simulated using ns2 [74]. Packets may be compressed at any node. Ignores decompression cost. Compression ratio fixed during simulation.
Xiao [75]	2010	None, RAR [76], gzip, bzip2, 7Zip. Whole file.	"Proxy" compression for email via custom client and server with communication side channel.	Client battery status, Signal-to-noise ratio (throughput), predicted energy use, CR (precomputed avg. by file extension) and user preferences.	Selected method with best "compression effectiveness" – a function based on monitors and precomputed tables.	Assumes decompression cost scales linearly by size.
Nicolae [77]	2010	None, LZO and bzip2 (one at a time). 64MB chunks in one experiment, 2MB chunks in another.	Cloud storage bulk transfer; compression added to the BlobSeer versioning distributed storage system. Supports parallelism and out-of-order operation.	Compressibility (64KB sample of 64MB chunk).	Compresses if sample CR beats a predefined threshold.	For reads, auto-decompresses if needed. Supports random access into compressed chunks (i.e., 64MB or 2MB) through middleware.
ACCENT [78]	2011	None, RLE [79], LZO, gzip, bzip2. Maximum TLS/SSL record segment is 16KB [80].	Modified OpenSSL layer within iCubeCloud cloud computing service. Backwards compatible with SSL/TLS.	Current bandwidth, CPU, expected CR and rate by algorithm (via training and periodic updating), and corresponding performance model of communication partner.	Periodically (here 200ms) chooses based on linear model of compression and encryption combinations (e.g., LZO + AES), parameterized by monitor data. Averages multiple rounds together.	Initialized via benchmark and, periodically re-tuned during use (200ms in their experiments). Memory sharing mechanism reduces algorithm switching cost. Relies on linear assumption from ACE [62].
CloudIO [81]	2011	None, QuickLZ-1 [82], QuickLZ-2, LZMA. Up to 128KB chunks including payload and metadata.	Designed for VM clients; integrated into Nephele framework [83].	Current and previous application data rate at MB/s level calculated every *t* seconds. (See note.)	Periodic hill climbing on monotonically strength-increasing methods with weights based on past success.	Project motivated by inaccuracy of CPU and I/O bandwidth reporting for VM clients. In experiments, adaptation period *t = 2* seconds.
DEEPcompress [84]	2012	None, gzip, bzip2, xz. 16KB chunks.	Dedicated test application.	Online LEAP [84] [85] energy consumption data (full-system or per-component).	Dynamic threshold model based on last observed energy costs for each method.	Designed primarily to demonstrate Online LEAP for dynamic decision making.

The purpose of most AC systems in the literature is to improve the throughput of data transfer, which saves time for an individual client (given otherwise identical situations). In terms of energy savings, Barr & Asanovic [3] have described how the “race to sleep” strategy and various algorithms would interact in their platforms (for example, how certain algorithms were able to use idle time more effectively). However, most AC systems that specifically target energy savings (particularly for mobile phone-like platforms) do this by saving time. In many cases, this is done by specifically interleaving receipt of data and decompression in order to eliminate idle time; they do not make any special decisions based on hardware idle time effects. Some systems explicitly mention saving storage space as a goal [72] [77], but this is along with, or as a secondary effect of, compression to improve throughput.

Again, these limitations are not necessarily shortcomings for their intended purpose. However, they do represent opportunities for further innovation, and because Datacomp is meant to be more general and less tied to hard-coded design choices than past systems, such limitations are important issues for my work to consider (which is not to say that Datacomp is unlimited or completely general, either).

In the following sections, I will discuss how my work relates to important prior work in adaptive compression and other related areas. In order to compare the various systems, I will discuss their similarities and differences within the context of the four AC components I identified in Section 3.3.3; Mechanisms, Methods, Monitors, and Modeling.

4.2.1 Mechanisms

Adaptive compression systems have been designed and tested in a wide range of application areas and abstraction layers. Perhaps the most commonly explored area is within

Grid and Cloud Computing [62] [64] [66] [77] [78] [81], where a large number of computers are combined in a distributed fashion. In this area, AC is primarily built into the middleware used by the Grid in question for exchanging data between nodes performing various workloads. Middleware approaches tend to benefit from the already highly abstracted design of the middleware (which hides internals from compute nodes), and from other system facilities (such as network monitors) that may be available to grid nodes. Middleware can provide AC benefits to client applications without modification and can also enable further efficiency through parallelizing operations or enabling out-of-order sending or receiving in a transparent fashion. Of course, typical end user systems do not benefit from this level of abstraction or infrastructure (although not all of the above systems require special grid facilities).

Adaptive Compression has also been explored as part of the networking infrastructure, separate from servers or clients [72] [31] [71] [73]. In these works, the researchers make adaptive decisions in some way and also consider special characteristics of the networking problem space. The specific applications in question improved the efficiency of network trace transmission and storage for network analysis, and also improved throughput in larger networks.

Mobile devices such as smartphones have also been a popular area for AC researchers [63] [68] [75]. In these works, AC is primarily used to improve the efficiency of network transmission in order to save overall energy spent. Systems in the literature include both Performance Enhancing Proxies (PEPs) [75] [63], where compression is performed on the proxy to save energy for the receiver, and standalone systems, which make decisions and perform compression or decompression locally.

Some systems are integrated at the application level or are built into networking infrastructure. DEEPcompress is a purpose-built AC network transfer application that makes decisions based on live energy monitoring using the Online LEAP energy measurement system [84]. NCTCSys is an AC system designed for improving the efficiency of transmissions of text data [59]. Perhaps due to its special purpose design, NCTCSys is not intended to be integrated into all applications or used in a transparent middleware, but rather would be integrated into applications that specifically process text data, such as web servers or email clients.

Adaptive Compression systems have also been built into general-purpose IO libraries or operating systems. Some systems are integrated into low-level systems such as the Linux kernel [55] or a Java Virtual Machine (JVM) [62]. Other systems utilize a library that typically exports a familiar interface, such as replacing the typical POSIX `send()` and `recv()` calls with adaptively-compressing analogues [66], or implementing AC within an SSL library [78].

In terms of messaging to communicate compression strategy changes, systems that always use the same compression method (e.g., IPZip [72]), or use a family of methods that are all decompressible by the same software (e.g., zlib as used in Knutsson and Bjorkman [55]) do not need a special messaging mechanism unless they are designed to also be compatible with non-AC systems.[18] Middleware systems often have their own communication protocol into which decompression information can be stored. Systems like ACE [62] use metadata inserted into the compressed data stream.

4.2.2 Methods

Methods describe the specific compression tools and parameters used in AC systems. Tools include the specific algorithms or utilities used, while parameters include "tunables" such as strength level, chunk size, whether parallelism was used, and so on.

[18] Nevertheless, Knutsson and Bjorkman implemented a simple negotiation mechanism (to enable or disable AC) using special TCP options [55].

4.2.2.1 Compressors

In terms of compression methods, four methods are highly represented in the literature: no compression, LZO, gzip, and bzip2. While these four are the most common, other algorithms are used in one or more systems. Some systems use QuickLZ [82] or LZF [67] as an alternative to LZO as a "fast compressor" and some use LZMA (xz) [22] as a "heavy duty" compressor in addition to or instead of bzip2. Furthermore, the general consensus is unified that these compressors tend to fill different niches in terms of compression power and computational requirements, with LZO being the fastest and weakest, followed by gzip which provides a nice middle ground, followed by bzip2 or LZMA which are strong, but computationally expensive.

In almost all systems (except where the data is almost certainly compressible [72]), the AC mechanism includes "null compression" – simply transmitting the uncompressed data. However, the specific collection of compressors used in adaptation varied widely across the systems. Not counting "null compression," some systems used only one method (sometimes after evaluating multiple methods) [68] [31] [72] [73], some used one method at a time in a series of experiments [63] [77], others used one method but adapted its parameters (such as strength) [55] [66], and the rest used a small number of methods [56] [62] [59] [64] [75] [81] [78].

4.2.2.2 Chunk Size

AC systems have approached the notion of chunk size, the discrete unit of adaptation for a given system, in a variety of ways. If the chunk size is too large there will be fewer opportunities for adaptation. If the chunk size is too small, adaptation will have a larger overhead because it will occur more frequently (and may pay some communication cost to switch techniques), and compression algorithms will tend to perform more poorly on the smaller inputs.

Nevertheless, most AC systems described here have a fixed chunk size (or fixed rule about input size) built into the compression algorithm or abstraction layer being used. So, while it may be transparent to the application, internally the system divides the input into smaller pieces for processing. The range of chunk sizes (for AC systems with a fixed size) extends from 16KB [78] up to 64MB [77], with most systems falling between 16KB and 128KB (see Table 10). Several systems did not have a fixed chunk size, but rather sent one file at a time, adapting with each new transfer [59] [56] [75] or compressed all data available for a given Opportunity. One proxy-based system chose which type of compressed data to send by adapting at the compressed block level (the discrete output chunks emitted by the compressor) [63]. One networking-oriented system proposed compressing network data on a per-packet basis [73], while a system designed for compressing network traces performed adaptation based on a configurable timeout value (which determined how much data was compressed at once) [72]. In the latter case, they described how (given perfect knowledge) they could solve an equation to identify the ideal timeout value.[19]

4.2.2.3 Pipelining and Parallelism

Virtually all AC systems stress the importance of pipelining compression and transmission through the use of separate threads. This way, the system can continue to compress or decompress data while it waits for data to be sent or received. This helps ameliorate some of the delay imposed by compression and adaptation.

[19] They also acknowledged that this time limit could be automatically controlled based on some model but declared this out of scope.

Truly parallelized reads and compression (i.e., multiple threads reading or compressing at the same time), when discussed, was typically not performed at the level of an isolated multi-core device as in Datacomp. Instead, it was provided via the use of grid middleware and redundant data across multiple nodes. For example, Nicolae used multiple hosts to perform parallel reads and writes [77]. Their middleware also supported out-of-order IO, enabling further performance benefits.

4.2.2.4 Strength Levels

While many LZ-family algorithms (such as DEFLATE and LZMA) can genuinely tune the amount of time or memory used in finding matches, the trade-off between strength level and results is not usually linear as might be expected [49] [50]. Furthermore, not all algorithms support real strength levels. For example, some algorithms provide tunable "levels" but which adjust other parameters (such as selecting slightly different algorithms as with LZO or adjusting the amount of memory used the case of bzip2). These "emulated" strength levels do not always translate directly to improved compression performance [9] [21] [86].

4.2.3 Monitors

Because the core tradeoffs in AC are the same, regardless of the design of a particular system, a core set of values are monitored in most AC systems. These include the unit times to transmit and receive data, the time cost of compression and decompression (for the given Event type), and the compression ratio achieved for a given operation. In this section, I discuss the properties that are typically monitored, and how the systems cited here accomplish that goal.

Some of the high-level properties mentioned above can be measured directly on the local side (such as compression time). Direct monitoring of certain types of information requires additional hardware. For example, energy models can be constructed relating time or observable events to energy consumption, but the accuracy of these systems obviously depend on the fidelity of the model (which may not be portable across different hardware platforms). However, even rough measurements of energy consumption (such as those obtained using the ACPI Smart Battery interface [87] or external tools such as a Kill-A-Watt monitor) require special hardware to be added or already be present in the system.

Obtaining highly accurate energy measurement requires integrating samplers and sensing leads to the test system, followed by software instrumentation allowing the workload results to be extracted from the sample data. One promising solution for this purpose is the LEAP[20] infrastructure developed at UCLA by Digvijay Singh, Dr. William Kaiser, and others [85]. This platform includes high-resolution component-level energy measurement (up to 80kHz) synchronized to specific events in software. While the original design of LEAP required data analysis to be performed offline, a new “Online LEAP” feature enables energy data to be available in near real time. Online LEAP data has been used in DEEPcompress, a proof of concept system driving AC choices using a simple dynamic threshold model [84].

Some properties are locally measurable in theory but not in practice, depending on the abstraction layer used by the AC system. For example, it can be difficult for a user-space application to get an accurate measurement of transmission time for a particular payload, because the real transmission times are abstracted away by the kernel. This can be seen by `send()`ing several kilobytes from an application on a slow network; if the send buffer is empty, the kernel will accept the data immediately. Thus, from the application’s perspective, the transmission time will be almost zero (meaning the available bandwidth is almost infinite).

As a result, properties must sometimes be estimated by proxy (i.e., by using some other measure to stand in for the true value) or through mechanisms with some inherent uncertainty.

[20] LEAP is referred to as DEEP in some documents.

Sometimes proxies or estimates are used not because the information is impossible to obtain, but because obtaining it is just too costly. For example, the compressibility of some data queued for transmission could always be measured by compressing the whole input; however this would be tremendously costly, especially if the data is uncompressible. Finally, in some circumstances, capabilities exist to provide monitored data to the AC system that normally do not exist in the classical operating system environment, such as the Network Weather Service used by ACE and DCFS [56] [62].

4.2.3.1 Communication Costs

The time required for clients to send and receive data is a critical measure in an AC system, because the time for a given communication operation is the "window of opportunity" in which compression must be profitable. Most systems model this as the available bandwidth or throughput of the channel. It is important to underscore that the important statistic is the *available bandwidth* – how much "space" there is in the network for the transmission in question. It does not help an AC system very much to know that it is using a 100-megabit data link layer if there is only one megabit available to the application. By considering available bandwidth, AC systems consider all the elements that ultimately determine how large the transmission/compression window is.

AC systems monitor available bandwidth (ABW) in a variety of ways. Systems that operate at the kernel level, use a Grid middleware, or have monitoring tool like the Network Weather Service (or are in some other custom environment such as network monitoring) can obtain bandwidth information directly from an existing internal facility or separate monitoring service [56] [62] [64]. Systems limited to more traditional sources of information have used various proxy values for bandwidth that trade off accuracy and availability; a somewhat fuzzy source of information that can be queried inexpensively could be more useful than an accurate method that is costly to use.

Examples of these "proxy values" include the signal strength of a wireless radio [68], the signal-to-noise ratio [75], line speed and application-level values such as the number of clients connected to the service [59]. One family of AC systems uses the fill and drain rates of a buffer to estimate whether the current compression strategy is improving or degrading performance [55] [66]. Still other systems have estimated or measured throughput at the user level with the simple arithmetic of bytes sent divided by the elapsed time [81] [78] [31] [66]. This approach is straightforward to calculate and accurate in the limit, provided that payloads are large enough or fast enough to flood the buffer. However, this "running average" approach can be problematic for short duration events and small payloads due to kernel buffering and management as discussed previously.

Some AC systems (e.g., many middleware approaches) have special channels or mechanisms for communicating non-payload information relevant to adaptation between parties [56] [59] [62] [64] [75] [78]. Without these channels, it is especially challenging for AC systems to model the performance of the receiver, which can be critical for obtaining maximum efficiency. While the overall throughput of the receiver can be estimated by the rate at which it consumes input from the network (similar to how we can estimate the outbound rate for senders), this information, like outbound rates on the sender side, is partly obscured by abstractions such as buffering provided by the systems on either end of the connection. This means that while the receiver can estimate its throughput and local conditions fairly accurately, it is more difficult for the sender to estimate conditions on the receiver side. This complicates matters since the sender is typically the system making compression choices. Thus, senders must be able to succeed using estimations of these values unless they specifically engineer a channel to transfer this information.

4.2.3.2 Monitoring: Decompression vs. Network

Even more problematic for AC systems without a means for passing adaptation information between parties, the standard networking paradigm does not tell a sender why data consumption rate at a receiver has changed. This is significant because it is not possible for a sender to know if a receiver's consumption rate has decreased because of network conditions or CPU load. If the decrease is due to the network, additional compression may be able to overcome the tighter bottleneck. However, if the decrease is due to high CPU load on the receiver, stronger compression could further degrade throughput [66]. While a messaging mechanism can help avoid this problem, I will also discuss how this potential "vicious cycle" (and a related controversy) can be coped with in the section on modeling approaches.

4.2.3.3 Compressibility

Most systems consider or estimate future data compressibility in some way. ACE [62] does this explicitly through what I refer to as the last block assumption (LBA). Under the LBA, the compression ratio of the previous input chunk (the "last block") is used as the estimate for the next input chunk. In this way, the LBA is a monitor predicting the future compression ratio. ACE uses this value directly in the process of choosing which of multiple algorithms to select. The LBA likely leads to small mispredictions when the data type changes, and would lead to larger mispredictions if the data type changed rapidly over a long period.

Other systems that explicitly consider the compressibility of the *current* input use sampling to drive compression decisions [64] [77]. In these systems, some small fraction of the available input (between 0.01% and 3% of the payload in those covered here) is compressed and the resulting sample CR is used as the compressibility estimate for the entire input. Sampling can be effective, especially if there is reason to believe that the overall compressibility of the data does not change often. However, if sampling is performed naively (such as by always sampling the beginning of a file where headers may be sparse) or if data compressibility changes rapidly and unpredictably (as it often does in the general computing environment), sampling can lead an AC system to make maladaptive choices.

Systems focusing on pre-compressed data know the compression ratio (or the compression gain in this case) so it need not be estimated [56] [63]. The other explicit approaches seen in these works use pre-computed averages in some way (depending on how many methods they support). One method is to assume that compression ratios for a given algorithm are stable for files having a particular extension (e.g., .doc, .pdf, .jpeg) [75] or for data sent via a particular service [73] (e.g., IMAP). However, this approach is not without controversy.[21] [22] While some files (e.g., MP3, ZIP or JPEG) are almost never compressible, the compressibility of files with extensions such as .doc or .pdf can vary widely based on their composition and origin. Also, in many circumstances (e.g., a network stream) the file extension may not be available and so cannot be used.

A final explicit method was employed in ACCENT [78], which uses a training phase to obtain the average CR for each algorithm used, irrespective of file type or extension. In this way, the CR is predicted as a function of the compressor. Additionally, in ACCENT (unlike other systems that used pre-computed averages) the average CR for each algorithm could be periodically updated via a feedback mechanism (every 200ms in their experiments). On the one

[21] "… [M]any file types are too general to make an accurate estimate. For example, binary files exhibit a wide range of compression characteristics which cannot be estimated using a limited number of samples." [57]

[22] "…[C]ertain types of files, for example, tar files, Power Point presentations, and PDF documents, may contain different types of data objects such as texts and graphics." [58]

hand, ACCENT's focus on one type of data (web traffic) and its periodic updating mechanism support the choice to ignore data characteristics and focus on compressor average CR alone. However, it is important to note that this will almost certainly result in a loss of accuracy (the system is unable to immediately recognize best and worst cases), and the overall accuracy of the design will depend on the workload and the adaptation period ("quantum" or "duty cycle" in their work).

Even if a system does not explicitly track CR [55] [59] [66] [68] [31] [81], most AC systems monitor at least one value that incorporates compression ratio in some way due to a simple passive mechanism: the effect of AC is at least in part dependent on the effectiveness of compression. Thus, assuming all other variables are stable, a degradation of compression ratio while compression is in use will cause a decrease in performance (in terms of whatever statistics are measured), which can trigger the system to adapt the compression strategy. By the same token, any time compression improves efficiency it is due at least in part to an acceptable compression ratio achieved by the algorithm.

4.2.4 Modeling

Models are the AC components that consume monitored values and make compression decisions. AC models analyze the monitored system state and attempt to identify whether, and if so, how a change in compression might improve efficiency for a given opportunity. One way to organize AC system models is to order them by the number of possible choices the decision mechanism can make.

4.2.4.1 Binary Choice Systems

Speaking generally, the simplest AC systems make a binary choice to enable or disable a single compression mechanism. In these cases [63] [68] [31] [73] [77], compression may be performed on the fly, or data may already be compressed and the AC system merely chooses whether to transmit compressed or uncompressed data. In any case, binary choices make intuitive sense; the model simply needs to determine whether the single compression choice available is a good idea for the given Opportunity.

This leads naturally to the use of a *threshold model* where compression is applied if some monitored statistic (either empirical or derived) is either above or below some value, and is disabled when the opposite is true. In the simplest cases (which are typically tuned for a given platform and workload), the threshold in question is predetermined and static, and the statistic is simply read from a system facility. For example, Maddah and Sharafeddine enabled compression when the wireless signal strength was below a certain level, and disabled it otherwise [68]. Many single statistics can be used to drive compression choices; for example, Nicolae enabled compression if the estimated CR (determined by sampling) was better than a certain level, otherwise not [77].

From a big picture perspective, fixed thresholds are only acceptable if there is reasonable confidence that the significant factors (e.g., platform, bandwidth, data type) will not change. Ideally, the choice of a threshold is determined not by hand tuning the system but by a dynamic mechanism incorporating system properties so that the threshold modulates to match changing conditions. Fully dynamic mechanisms for binary choice models appear to be rare, although several projects took steps towards this design. Xu precalculated a threshold using equations (described in the paper) relating the relevant monitored conditions [63]. The mechanism did not need to be dynamic because conditions did not change during the evaluation. Rather than producing a threshold for comparison against some external value, Shimamura compressed when an equation model using multiple monitored values was satisfied – although the simulated evaluation ignored decompression cost and assumed a fixed compression ratio (based on precalculated averages by simulated traffic type) [73].

A family of binary choice AC systems [31] [71] uses a dynamic mechanism that is superficially similar to Knutsson and Bjorkman [55] and AdOC [66] (early AC systems described later). These systems use the state of a network pipe as a mechanism that drives compression choices. Like IPZip, these systems always perform compression. However, while IPZip chose when to *re-train the preprocessor*, these systems buffer data and adaptively determine when to *execute compression* and transmit the result. In these systems, compression is performed (with a single algorithm) if the network buffer is full or if there is some data buffered and a timeout elapses. In Politopoulos [31], this timeout (which is a kind of threshold) is dynamically computed based on the inter-arrival time and average throughput.

Finally, a slight variation on the binary approach is IPZip [72], which always compresses (with gzip, in this work) after applying a special-purpose preprocessor for network data (IPZip). Instead of enabling or disabling components when observed CR degrades, IPZip *retrains* the preprocessor on the new data. This is still a binary choice – “keep the current IPZip plan” or “use a new IPZip plan”.

4.2.4.2 Higher-dimension Modeling

Moving upward from binary choices, AC systems can choose from three or more methods, one of which typically implements “null compression” (i.e. sends raw data) while the other methods are meant to offer various cost/benefit trade-offs. The compression choices may all be discrete algorithms or libraries [75] [78] [62] [84] (such as a system using DEFLATE, RLE and bzip2), from the same algorithm with varying parameters [66] (such as gzip levels -1 to -9), or combinations of both [64] [66] [56] (such as gzip -1, gzip -9, and bzip2). Often, authors explicitly attempt to provide a range of compression choices from “fast-but-weak” to “strong-but-slow”. Whether this hierarchy is rigid or flexible varies from system to system.

The systems surveyed here support between four and eleven compression methods. Systems on the higher end of this range (the related Knutsson & Bjorkman [55] and AdOC [66]) leverage the strength levels provided by gzip, with the explicit assumption that increasing the strength level increases potential compression (an assumption that does not always hold). Systems on the lower end of this range choose between three and five hand-picked methods.

Simple threshold models are sometimes used for these higher-order systems as well. However, to select the right method, rather than simply whether to use *the* method as in binary systems, the model defines ranges for each method based on various empirical or derived statistics. NCTCSys [59] is a higher-order AC system that bases decisions on a precomputed threshold model and the run-time value of several monitored variables. DEEPcompress uses a simple history-based threshold mechanism. In it, thresholds are set based on the energy spent when each method was last used (measured and reported by Online LEAP) [84].

However, most higher-order AC systems base their models on a combination of equations relating monitored values with dynamic thresholds and some fixed precomputed information. Wiseman [64] used a fixed hierarchy of algorithms selected based on dynamic thresholds dependent on system conditions. However, most systems precompute difficult-to-obtain information based on statistics acquired during operations on training data, such as average compression [64] or decompression [56] rates by algorithm, expected compression ratio by filename extension [75], or linear models relating algorithm performance [62].

This can result in effective systems, although it raises an important design decision. “Baking in” the precomputed data permanently enshrines assumptions about conditions that may not always hold in uncontrolled circumstances. On the other hand, continually re-evaluating this information adds overhead to the system. Most systems that support updating this information attempt to strike a balance of some kind.

For example, Xiao's Performance Enhancing Proxy (PEP) for email system [75] used an equation to determine the "compression effectiveness" for each possible method. This value depends on both dynamic monitored values and precomputed tables of CR by file extension. At each opportunity, compression effectiveness is calculated for each choice (Xiao supported five methods including "none"). Then, the method with the highest "compression effectiveness" would be used to compress the data before sending it to the client. Using equation-based models in this way avoids a strict hierarchy of methods, but requiring calculations to evaluate each method at adaptation time results in more overhead which can limit the effectiveness of the system or the number of methods considered.

4.2.4.3 Knutsson and Bjorkman and AdOC

In 1999, Knutsson and Bjorkman's Adaptive Compression project [55] implemented an adaptive network compression scheme to improve performance in terms of time, and found their system generally able to adapt to changing characteristics and benefit over a non-adaptive strategy.[23] Their design set the zlib compression level based on the length of the Linux kernel's TCP write_queue – the buffer of data waiting to be put into the send buffer. Since their system was built into the Linux kernel, applications did not need to be modified in order to benefit from it (although the receiver needed to be compatible to perform decompression).

[23] As with other systems, I am avoiding discussing specific performance results or limitations (e.g., at what bandwidth the system failed to improve efficiency) since these are heavily dependent on the hardware available at the time, the test conditions, and the data used.

Their scheme monitors the length of the queue, and using a built-in function, maps the queue length to a zlib strength level. Longer queues were mapped to higher compression, since queued data indicates a bottlenecked network. Conversely, shorter queues were mapped to lower compression levels (including no compression). In order to reduce fluctuations in strategy choice, they commit to a method choice for a number of packets unless the queue length changes "drastically." I refer to this mechanism as "lock-in." They also flush the compressor following a write to ensure that no data is trapped waiting for compression in the buffer. Ultimately, their project demonstrated adaptive, transparent compression in the kernel using a variety of file types, CPU loads, and network conditions.

In addition to improving throughput in many circumstances, they described many effects found or discussed by future AC designers. They found that decompression was cheaper than compression in their tests, and that decompression cost can be inversely proportional to the compression cost. In other words, the harder a sender tried to compress while sending, the less expensive the receipt and decompression was for the receiver. This suggested that stronger compression could be worthwhile not just for low-bandwidth networks, but potentially (and somewhat paradoxically) when the receiver is suffering from CPU load.

They also noted that oscillations in method choice often occurred due to their fixed mapping of queue length to strength level – a common problem in AC systems. While they chose to address this by weighting the current choice, they described how mitigating this issue has its own pitfalls. For example, using smoothing techniques (such as rolling averages) to reduce oscillations resulted in missed opportunities when the network changes fluctuated rapidly. They also investigated the effect of their adaptation system on network delay.

Jeannot, Knutsson, and Bjorkman's "Adaptive Online Compression", or AdOC [66], was an expansion of the system by Knutsson and Bjorkman. They used a user space library and FIFO rather than making kernel modifications, and provided a much more dynamic model meant to cope with the shortcomings of the earlier system. In particular, the change of compression level in AdOC is related most directly to the *rate of change* of the queue length, not simply the length of the queue. This allows for compression to be "ratcheted up" when the queue grows longer (and increased faster when it grows more rapidly) and vice versa when it grows shorter. This allowed their model to break away from a "length to strength level" mapping and instead use relative performance to control adaptation. Since they used a user-space mechanism, applications needed to be modified in order to use AdOC. However, they provided a library with function calls offering extremely similar semantics to system calls like `send()`, `recv()`, and so on, so modifications were minimal.

They also recognized a pitfall in many AC systems, which can occur when the compressing side is unaware of the decompression cost on the receiving side and assumes that compression improves throughput. The problem is that throughput can decrease because of network conditions, *or because of decompression cost on the receiver*. In the standard network communication paradigm, there is no communication channel from the receiver to indicate its system load or compression preferences. Thus, if a system increases compression and throughput degrades (because of decompression time), the sender may continue to increase compression, further degrading performance.

While decompression is *often* faster than compression, and stronger compression can *sometimes* result in faster decompression, this is not guaranteed, especially when systems are mismatched (e.g., a server and a smartphone). To cope with this pitfall, AdOC monitors whether compression increases are actually improving performance. If they are not, stronger compressors are blacklisted for a period of time (one second in their paper) in an attempt to “wait out” the performance degradation.

Finally, AdOC included many edge cases in their model in order to cope with special circumstances. For example, they included the high-speed algorithm LZF [67] (similar to LZO [21]) for high-speed networks. They also determined size and throughput limits after which point AdOC could not improve efficiency. They also tested the effect of compression on buffered data (a type of sampling) and disabled compression if the data appeared to be uncompressible. They also investigated their system’s effect on delay.

4.2.4.4 ACE

Krintz and Sucu developed ACE [61] [62] in 2003, which is an intellectual descendant of DCFS [56]. Built into the ORP Java virtual machine [88] rather than a kernel or a library, ACE is an adaptive network compression system which chooses a compression algorithm and parameters based on many factors, including internal compression cost-benefit models, compression ratio comparison models, dynamic estimates of the compressibility of the data stream, and external predictions of CPU and network utilization, provided by the NWS [89]. Because ACE is implemented inside a Java virtual machine, network applications written in Java can benefit from ACE without any modifications (although once again the receiver must be compatible).

ACE introduced a number of significant advances. First, ACE chooses from a selection of discrete algorithms (none, LZO, zlib, and bzip2), rather than simply adjusting the compression strength of one algorithm. This can offer a more significant performance variation than simply adjusting the compression level of zlib. ACE considers many system parameters, including compression ratio (using the “last block” assumption), system load for both the sender and receiver, network conditions, and more.

Many of these values are provided to ACE via the Network Weather Service (NWS) [89], a system and network resource monitor and prediction framework. This provides ACE nodes with special knowledge about their peers and network conditions that legacy systems do not have. ACE also considers the history of resources such as the past behavior of sockets to help estimate various parameters, and, like AdOC, also uses buffer fill and drain rates as a proxy metric for data throughput.

One of ACE’s major contributions to AC is its novel compression performance prediction mechanism. In addition to monitoring internal system characteristics and using data from NWS, ACE chooses which method to use by combining the “last block” CR estimation monitor with a novel model that predicts the compressibility and performance of one method given the performance of a different method. In short, the authors of ACE identified that, with respect to their test data, the compression ratio and run-time performance of each of their compressing methods (LZO, zlib and bzip2) could be related to one another using a linear model.

By building up a precomputed model using training data, they constructed linear relations for each pair of algorithms. With these fixed models, the CR of the previous block as achieved by the previous compressor and the system properties ACE monitors at run-time, ACE could “look up” the predicted performance for the next block for every other compressor. ACE then picked the method that the model predicted would have the greatest efficiency for use with the next block.

4.2.4.5 ACCENT

A dynamic and equation-driven model that chose between multiple methods for on-the-fly compression was presented in ACCENT [78] (2011). ACCENT explored the energy costs associated with combinations of compression methods and SSL ciphers for web traffic under changing conditions. Comparing combinations of compression and encryption was done based on the idea that encryption formed a competing computational workload that might affect the best compression choices. By integrating their system into an SSL implementation, applications that used SSL did not need to be modified to benefit from ACCENT.

Their model relied on the linear relationship between compressors described in ACE [62], which suggests that for a given input, the difference in achieved CR and compressor run-time across multiple methods can be related by a linear (or near linear) relationship. In other words, given the performance of one algorithm in an Opportunity, it should be possible to predict the performance of a different algorithm in the same Opportunity. Using this assumption (and SpecWeb2009 training data [90]), ACCENT created equation-based "floating scales" which derived a single performance value, called the "computation index" (CI) (similar in spirit if different in detail to Xiao's "compression effectiveness") for each possible method+cipher combination. Then, every time quantum (or "duty cycle"), which was 200ms in their paper, the relative values of the CIs would be adjusted (or "floated") based on the linear assumptions from ACE and the other monitored conditions. While the CR values incorporated into the floating scales were created through training, they could be updated with real history data over time as well.

ACCENT made compression decisions by consulting the current floating scales of both the sender and receiver at every time quantum, which were transferred between hosts using a special mechanism. This allowed decisions to be made based on the needs of both parties and allowed the models to update themselves over time. ACCENT's models are simple, easy to understand, and powerful.

4.2.4.6 CloudIO

A simple and extremely dynamic model is demonstrated in Hovestedt's CloudIO [81], which was designed for Virtual Machine (VM) guest hosts. CloudIO copes with the fact that CPU and I/O bandwidth reporting on VM guests is often unreliable by basing its adaptive model strictly on the current and previous application data rate. This rate is calculated on a regular basis (every two seconds in the paper). Cloud IO chose between no compression, QuickLZ-1 [82], QuickLZ-2, and LZMA [22] and, due to its long adaptation period, adapted over megabytes, rather than kilobytes (an explicit design choice meant to overcome network fluctuations). In addition to not needing CPU or I/O-level bandwidth information, CloudIO does not require any training for compression ratio or other models. Instead, CloudIO uses a kind of hill climbing approach with various weights and cooldowns.

4.3 Discussion

The previous AC systems each explore a different facet of the problem space and have made contributions to the state of the art. However, Datacomp is a novel and important addition to these prior works for three general reasons. First, many prior systems rely on hard-coded assumptions. Sometimes these assumptions are simplifications made to facilitate research (such as assuming that CR is fixed for a given traffic type [73]), while in other cases they are propositions that are relied upon as a ground truth but which may or may not hold in a broader environment (such as the linearity assumptions used by ACE and ACCENT [78] [62]). Both kinds of assumptions can limit the generality of the system.

Second, most of these systems were not designed to succeed in nor were they evaluated for the uncontrolled general computing environment. This is not a flaw of the prior systems, but it makes it difficult to extrapolate the effect of their performance on general-purpose systems. Some systems are clearly special purpose or are proofs-of-concept and simply do not apply to general-purpose computing, even if they are interesting and successful AC systems [68] [72] [73] [31] [71]. But several of the more complete systems target Grid or Cloud environments, which often support the use of external facilities like the NWS [62] [89], built-in parallelism [64] or communication channels for exchanging data that do not exist in the "real world" [59] [62] [78].

Similarly, evaluations that target special environments (e.g., disk images in a grid computing environment) or use a limited set of test data (e.g., one file of several types repeated to reach a total byte count) are not always very representative of the data and environments encountered by typical users [62] [66] [81]. In particular, the workloads for many of these systems were proportionally much more compressible and repetitious than typical user data. This is partly because the standard compression corpora (the Canterbury and Calgary Corpora[24]) are widely known and available but are not especially well-rounded or representative of user data; instead they are meant for evaluating compressors in terms of CR. Unfortunately, reliance on specific or limited forms of workload data can make it difficult to extrapolate the generality of the models and results.

4.3.1 Zlib and LZMA Strength Levels

Future AC systems should carefully pause and reconsider relying on LZ strength levels (i.e., gzip-1 through gzip-9 and xz-1 through xz-9) alone to provide a spectrum of meaningfully different compression choices as opposed to selecting between various algorithms. Results in

[24] http://corpus.canterbury.ac.nz/

this work (Sections 8, 9 and 10) show that different discrete compression methods, such as lzop, gzip and xz, tend to provide more diverse performance (in terms of compression ratio and computation time) than simply increasing the strength level of zlib or xz.[25] Furthermore, compression ratio may not even strictly improve as compression level increases for all input (see Section 8.4). These and other issues suggest that single-algorithm compression levels are not the most effective "knobs" for an adaptive compression system.

4.3.2 Adaptation Quanta

Another serious issue for my purposes is that most systems use only one chunk size (or adaptation duration), and this setting appears to be chosen either arbitrarily or empirically based on the test data at hand. A single quanta limits the situations where AC can be profitable. For example, ACE does not compress if the input is less than 32KB [62], and AdOC does not compress less than 80KB [66], which would immediately preclude compression for many operations that users perform in day-to-day tasks, such as downloading many small components of a web page. Other systems commit to one strategy for long stretches.[26] I believe these choices were made because many systems rely on long-term sustained throughput with only moderate variation. However, these limitations would result in missed opportunities altogether (when the data is too small) and in suboptimally adapted periods (when a long commitment precludes a more fine-grain strategy) in uncontrolled environments. Fixed long adaptation periods also increase the cost of poor choices, increasing the penalty to the overall system's efficiency.

[25] There are cases when increasing the strength level is very significant, as discussed in Section 8.8.
[26] For example 200ms (about 2.5MB on a 100-megabit network) in one system [76], and 2MB or 64MB chunks in another (fixed based on experiment conditions) [78].

In contrast to these approaches, Datacomp is able to select different chunk sizes (ranging from 32KB up to 512KB) at runtime, and will attempt to compress payloads as small as 1.5K[27]. Datacomp supports these options because my data shows that the size of the input can have an impact on the optimal compression choices (see the chunk size graphs in Section 9.4) and that compression ratio by input size is not consistent across multiple algorithms (Figure 27 through Figure 30). Thus, as input sizes change, the relationships between CR and algorithm can also change. However, small adaptation quanta generally result in less “profit” due to increased overheads and decreased benefits. As a result, in order to make these kinds of choices and be profitable, the AC system must be flexible and efficient overall, and should be evaluated in many contexts.

4.3.3 Linear Relations Between Compressors

Given a set of known information, an AC system needs to predict what method to apply (the unknown). ACE meets this need with linear models, created offline, of compression and decompression performance averages for each machine under test. ACE's algorithm comparison tables use a linear model built by correlating the compression ratios (and runtime performance characteristics) achieved over 32KB segments extracted from a mixed training set of files between pairs of gzip, LZO, and bzip2. When ACE compresses a block, it consults the model to predict how well an alternative algorithm would have compressed the same block, and how long the operation would take. While compression ratio should be the same from machine to machine, runtime performance characteristics depend on the local device and are generated by a calibration phase during the installation of ACE.

[27] This value is configurable.

The authors of ACE described experimental measurements showing that the CR and run-time of zlib, bzip2, and LZO on a given set of training data were approximately linearly related[28]. In other words, if zlib's CR (or run-time) is worse by *X*, then LZO's CR (or run-time) is likely to be worse by *Y*X*, where *Y* is some coefficient relating the algorithms and property in question. As a simple example, suppose that ACE has been compressing a stream of 32KB blocks with LZO, and that in the comparison model, a CR of 0.9 for LZO and a runtime of .01s (given a particular 32KB block) is strongly correlated to a CR of 0.5 and a runtime of 0.2s for bzip2. While preparing to compress the next 32KB segment, suppose that ACE learns from NWS that there will probably be enough CPU time during the next interval to use bzip2 without a latency penalty. Then, informed by the model, ACE expects bzip2 to achieve a CR of 0.5 for the next 32KB segment.

ACCENT explicitly relies[29] on ACE's assumptions. ACCENT uses this assumption as one of the foundations of the floating scale design, since this allows the scales to be "floated" in relation to one another based on observed performance (since performance is assumed to be linear), rather than be completely reconstructed. Relying on these assumptions is not necessarily a flaw in ACCENT or ACE, as the assumption is used to great effect. Rather, the potential issue is that while these assumptions clearly hold in many circumstances, they were derived in specific circumstances and have not been proved true in general. This could result in losses for an AC system operating in the uncontrolled general computing environment.

[28] "The results indicate that compression ratio and compression time as well as compression ratio and decompression time exhibit a near-linear relationship for each of the compression algorithms we investigated. Thus, we generate a linear regression for compression time and ratio and decompression time and ratio, similarly to our regression lines for predicting compression ratios across compression algorithms. ACE uses this function to efficiently approximate compression and decompression time from compression and decompression ratio at runtime." [57]
[29] "Despite the algorithmic differences, we assume an approximately linear relation among the different compression methods in terms of the CR and process speed [Citing ACE]. The [floating scale] can therefore be updated on the basis of the [computational index] of the current encoding scheme." [76]

First, ACE derived this assumption from its use of zlib, LZO, and bzip2 on a specific set of training data. ACCENT uses the assumption from ACE and builds its own model, but assumes that the linear relationship also applies to their RLE algorithm. However, it is plausible that certain data might not compress well using RLE, but would compress well using an algorithm like bzip2. In cases like this, the linear assumption (or the coefficients used to model the relationship) may not hold. In fact, in my own evaluation I ran into situations where one compressor (LZO) performed significantly worse than other compressors for specific data types (See Section 8.4).

I take the legacy of these projects as a starting point. They indicate that the optimality of a particular compression choice is based on a diverse set of factors, and that even simple heuristics can significantly improve performance. However, my own research shows that these heuristics may not hold in more uncontrolled environments, but that they can be improved upon – in many cases, significantly.

5 Comptool

5.1 Introduction

Comptool is an analysis tool for compression that identifies strategies that are close to the ideal for a given scenario – a set of input, bandwidth limitations, concurrent system workloads, and compression choices. Comptool does this using a benchmarked brute-force approach – trying every option available and then analyzing the results to find the best solution for a given set of priorities. Comptool then identifies the best strategy from these results using a greedy algorithm.

In addition to being able to identify the best strategy for time or space, Comptool is LEAP-enabled, [30] allowing it to directly measure the energy consumed at the component level during compression operations, which in turn allows it to identify the most energy-efficient strategy as well. In this section, I will describe the motivation, design, and use of Comptool, in addition to some discussion of its limitations.

5.2 Motivation

The primary motivation for Comptool is to provide a means of identifying the potential savings available from any compression method in general, and from adaptive compression in particular, given a set of input and environmental conditions. While the concept of Adaptive Compression (AC) makes intuitive sense, this information is valuable because it helps to establish whether the potential benefits of AC justify its development and use.

The value of non-adaptive compression is obvious and can easily be demonstrated. Compression is already widely deployed, using static approaches. Users are familiar with and

[30] More detailed information about LEAP measurement is available in 5.3.1.

use compression utilities, and many types of common files have compression built into them, especially files that are likely to be large, such as graphics, music, software archives and productivity documents. A reasonable amount of network traffic is also compressed; in particular, most webservers already perform DEFLATE compression on outgoing data (included as an option in the HTTP standard since at least 1996 [51]). There is typically little to no benefit in compressing already-compressed data, because most of the redundancy in the data has already been removed. As a result, simply performing more compression on top of the status quo is unlikely to be successful.

It is probable that there are classes of data that are not currently being compressed but that could benefit from compression. However, the total value of compressing this "low hanging fruit" depends on how much of it exists in proportion to data that is already compressed or is not compressible.   If low-hanging fruit composes only 10% of the existing data, then the additional benefit of the new compression strategy is limited to however much compression the new data will admit. In other words, if you save 30% on a class of data that only comprises 10% of the total, you will only save 3% of the total.

The more significant promise of a tool like Comptool is that it might be able to identify more effective compression strategies that could be used *instead* of the current methods. For example, it would be extremely valuable if Comptool identified that web data could be much more effectively compressed using some algorithm other than DEFLATE. The challenge facing this promise is that many people already do test compression methods for given workloads and are already thinking about how to squeeze a few more percent out of their systems. Thus, it is fair to be skeptical about the size of improvements that are possible.

Furthermore, this research is primarily about Adaptive Compression. Adaptive Compression is only worthwhile if the best strategy is not simply a *different static strategy*, but one that actually *requires* an adaptive mechanism. Since we cannot answer, in general, whether there is any better static strategy or what its benefits might be, we also cannot answer whether that strategy requires adaptation in order to maximize efficiency. These questions are of primary importance for applied compression and Adaptive Compression research; obtaining satisfactory answers to them are of prime importance for this work.

Comptool provides answers to these questions by exhaustively searching for the best compression strategy for a given scenario and goals. This information can then be compared against the performance of common methods such as "null compression" or DEFLATE to indicate whether any strategy beats the current status quo. These results also show the details of each adaptive strategy, including sequence of methods it is composed of (which reveals whether the strategy is static or dynamic) and the savings of the best strategy versus the alternatives.

Comptool functions by compressing the same data using a configured set of methods in a controlled execution environment while also recording a wide variety of performance statistics. These statistics include not only run time and space saved, but also the energy used (when using LEAP), CPU utilization, file type, and more. This wealth of data enables a fine-grained analysis and comparison between multiple compression methods, which is useful for those researching compression, developers comparing compression alternatives, and of course for the design and evaluation of adaptive compression systems like Datacomp. Since Comptool simultaneously measures elapsed time, space in bytes, and energy consumed, data generated by a single run of Comptool is useful for identifying strategies for different priorities. For example, after a single test sequence, analysis of the data allows users to identify the best strategies for time, space, and energy – without having to perform separate experiments for each set of goals.

When analyzing for space, ties can be broken in favor of time and vice versa. This means that the best strategy for time automatically includes space as a secondary priority, and the best strategy for space automatically includes time as a secondary priority. This means that secondary priorities do not need to be explicitly requested.

Comptool is also a powerful and multi-purpose tool for comparing various compression algorithms in a head-to-head fashion. Since all implemented strategies are tried separately, Comptool is not only able to find the best and worst strategies, but also enables the analysis of hypothetical strategies that might not be best, but could have other interesting properties, such as the best strategy that does not include some option X, or the best strategy for space that completes within some time limit Y. With multiple repetitions of the same test or careful modulation of test parameters, data generated by Comptool can be used to identify interesting results with strong statistical significance, such the average performance of a particular method in a particular scenario or the extent to which the best strategies for time and energy savings differ.

Comptool also has applications outside of research. It should be a useful tool for compression algorithm designers or developers interested in using compression in general, even if they are not planning to use an adaptive compression scheme. As a standalone tool, Comptool does not require a remote host or other infrastructure to run tests.[31] With virtually any input, developers can perform local experiments with Comptool in a rapid and convenient fashion. This means that developers can use Comptool to analyze the compressibility of the data emitted by their applications, informing their choice of compression strategy. A developer of

[31] Use of LEAP energy measurement is optional because it requires additional hardware.

productivity software could use Comptool to identify which compression library would provide the best compression ratio for documents (in addition to the time cost of that choice versus the alternatives), while the developer of web or mobile phone applications could identify which compression strategy would be best for saving transmission time or energy. Compression algorithm designers could use Comptool as a testing framework to compare the performance of their software by pitting their algorithm and its various options against other choices in a wide variety of ways.

Comptool is not limited to typical file data. It can be used to optimize compression for networking or network applications by compressing network traces as input. Network traffic can be captured live using a tool such as tcpdump [91] or Wireshark [92]. After extracting the payload data from the trace using a tool such as chaosreader [93], a user can send this data through Comptool and observe how it performs under various compression strategies.

For the purposes of this research, Comptool does most of the above. But perhaps most importantly, it provides the “ground truth” against which the performance of Datacomp is compared. Since the best strategies for compression are generally unknown, it is difficult to judge Datacomp’s performance except by comparing it to the status quo. By using Comptool to identify the best average performance of all strategies (adaptive or static) for a given set of scenarios, and then running Datacomp on the same scenarios, we can determine how close Datacomp’s performance is to the ideal. Further, by looking at the specific choices made by Comptool versus those made by Datacomp, we can help identify how and where the AC system made poor decisions.

5.3 Design

Comptool is a command-line tool written in Python, using the widely-available libraries liblzo2, zlib, libbzip2, and liblzma5 (the software underlying the compressors LZO [21], gzip [8], bzip2 [9], and xz [22]) to perform single-threaded compression. A user runs Comptool by providing input and setting a number of command line options controlling test parameters, logging, and the execution environment. Normally the results are automatically stored in an SQL database when the test is complete and offline analysis tools written in R[32] are used to identify the best choices based on the desired outcomes.

Comptool supports many options, but the most important ones are:

- **Compression methods**: Comptool tries a variety of algorithms configured with different options in the specified environments. These algorithms, along with the set of different input sizes to try (the "chunk size" or "quantization levels"), are defined in the source code but are very easy to modify. Like Datacomp, Comptool standardizes the interfaces for all compressors using a basic API. As a result, adding new compressors that already have a Python library to Comptool simply requires writing a Python wrapper to translate the Comptool API instructions for the underlying compression library.
- **Bandwidth limit**: Comptool can limit the rate that input is provided to the compressor and the rate at which output is transmitted over outgoing I/O channel. Limiting the output channel models the network bottleneck existing between most client and server relationships on a network. An inbound limit can also be imposed, limiting the rate at which data can flow into the compressor (modeling a slow disk, for example).

[32] http://www.r-project.org/

- **Contention workload**: In order to simulate other processes on the system contending for system resources, Comptool can start and stop parallel workloads that run during the tests.
- **Input data**: Comptool accepts the names of one or more files to process in sequence. Each file is processed using all methods at all quantization levels, with the resulting performance of all events written into an SQL database, which is later used to help isolate the ideal strategy.
- **LEAP sampling**: Comptool includes LEAP energy measurement tools, enabling it to accurately measure the energy used during compression operations.

Other important parameters include the CPU frequency, which can be set to a specific value (required for LEAP) or left to be dynamically controlled by the kernel. More options include whether and how to save data to SQL, whether to enable or disable LEAP energy measurement [85] (since most systems do not have LEAP capabilities), and how many repetitions of each task to execute.

Early versions of Comptool accepted resource priority values (numbers that specified the rankings of the resources “time”, “space”, and “energy”) as described in my prospectus [94]. However, as described in Section 5.2, it later became clear that specifying priorities at run time (to start dedicated runs for a specific goal) was not necessary, because Comptool could measure all relevant resource statistics for a given environment at once, with analysis for a given set of goals being performed offline.

5.3.1 Compression Methods

When a Comptool test is run, each distinct compression Method supported by the installation is used on each chunk of input Data in the controlled Environment. In other words, Comptool generates all possible Events for the current configuration (including the set of chunk sizes to use). Comptool currently supports the same libraries underlying Datacomp. While the set of algorithms used is not configurable at run time (due to the complexity of specifying such a list), they are defined in a list of options at the top of Comptool's source code. Thus, if a user wishes to use different algorithms, or provide different options, he or she can easily do so.

I originally intended Comptool to use command-line utilities to compress the data. Using standard compressors such as the utilities lzop, gzip, bzip2, and lzma would make Comptool's results understandable to virtually any interested party. These utilities are also generally written by experts (often the algorithm designers themselves or library implementers). Because of this, the best set of options and parameters (e.g., strength levels, memory usage) for the underlying libraries are used, which may not be known to casual developers. Since the utilities are widely distributed this creates a *de facto* standard for how users expect the compression utility to perform. They are also open source (unlike RAR [76], for example), so it is likely that many users and developers have investigated and optimized the code against the limitations of the library.

Because I used file utilities to provide compression, I was able to use the Datacomp-enabled dzip (described in Section 5) to provide multi-threaded compression. Standard utilities (e.g., gzip) were used to provide single-threaded "classic" compression, while dzip was used with threading enabled (e.g., dzip using zlib strength 1 with two threads enabled). This allowed the comparison of threaded compression performance in Comptool solutions.

However, while the utilities are written in a compiled language, their nature as standalone file utilities imposed unanticipated costs due to starting and stopping separate processes, the scheduling overhead introduced when separate processes need to communicate with each other, and so on. As an example, executing even the simplest process (e.g., /bin/true[33]) on my test systems requires about 0.003 seconds. Starting a compression utility would take at least this long, but probably significantly longer since the compressor performs meaningful work.

Unfortunately, 0.003s is too long of a delay for many of the environments I was testing. For example, at 100Mbit/s, 40K of data could be transferred during a delay of 0.003s. This overhead made the compression operations appear to take much longer than they would have in a real environment where data wouldn't be quantized or where compressor costs would be more reasonable. This ultimately limited the environments that Comptool could faithfully emulate. These limitations could be somewhat mitigated but they could not be completely eliminated – the cost of using starting a new process for each chunk of data was just too high.

As a result, I moved Comptool to a library-based approach. In this scheme, the chunk data is read into memory before starting the timer. Once the timer is started, compression is performed in memory, and the buffer is written to disk using a bandwidth limiter as an intermediary. This is much faster from a performance perspective than file utilities (by an order of magnitude in some cases), allowing more accurate measurements of shorter operations and thus more faithful emulation of environments with faster networks.

Using libraries in serial within Comptool is less transparent to users of Comptool than file utilities, because libraries can have many options available at run time, while common utilities somewhat "standardize" behavior as just described. Using libraries creates the requirement that API wrappers must be written and used, where command-line utilities could simply be called (already having a uniform API). Using libraries also unfortunately eliminates the potential beneficial effect of parallelizing compression and transmission through the use of separate

[33] "true" is a utility for which the description is "do nothing, successfully". true does nothing and quits with a status code of "0" (which is logical "true" in shell speak). In contrast, "false" is said to "do nothing, *unsuccessfully*". It simply exits with the status code for failure (1).

processes. However, for Comptool this was not a large effect as the parallelism benefit was attenuated by the cost of scheduling and inter-process communication.

Using libraries also meant that I could not use dzip to provide threaded compression without creating a Python interface for Datacomp. However, since standard compressors do not provide parallelism (and Datacomp still does), I felt that it was acceptable to limit Comptool's results to single-threaded solutions. Ultimately, the overall accuracy improvement enabled by libraries made this approach worthwhile. Nevertheless, this design choice is an example of the challenge of building an AC oracle and the kinds of competing values in Adaptive Compression systems that cannot be completely reconciled.

5.3.1 LEAP Energy Measurement

Modern computers do not have any hardware capable of performing consistent and accurate high-resolution energy measurement at the component level. The typical approaches to meet this need are to rely on the Smart Battery interface included in many devices, to rely on an external tool such as a "Kill-A-Watt" device, or to build a bespoke energy measurement system.

The Smart Battery (SB) interface is designed to provide a "fuel gauge" for laptops, not to support high-resolution energy experimentation. It uses the SMBus, which for most devices as of this writing supports a maximum data rate of 100K/s. A recent project calculated that the battery interface could only be polled effectively at a maximum rate of once every 8.79ms, which translates to a minimum sample time for the battery interface of 0.00879 seconds [87]. On a 100-megabit network, approximately 100K can be transferred during this time. Using a household energy measurement tool, such as a Kill-A-Watt[34] (which must be read manually) or a

[34] http://www.p3international.com/products/p4400.html

WattsUp?[35] are likely to have even lower resolution. For example, the WattsUp? PRO device has a maximum sample rate of 1Hz [95]. Additionally, none of these systems are able to resolve the energy consumed at the component level, which could be important to know. For example, component-level measurements are necessary to capture the energy variation of the RAM or CPU caused by different Events.

While various projects have created their own energy measurement platforms over the years by combining measurement hardware, workloads, and analysis software, the effort of creating such a system is substantial. Furthermore, the capabilities of the resulting systems vary widely in terms of synchronization, number of channels, sample rate, difficulty of implementing, overhead of instrumentation, and more. Fortunately, it is no longer necessary for researchers interested in high-quality, high-resolution component-level energy measurement to build their own systems, thanks to the LEAP energy measurement system developed by Digvijay Singh and others from Dr. William Kaiser's lab in the Electrical Engineering Department at UCLA.

While complete details may be found in our whitepaper [85] and on the project's wiki,[36] LEAP is a complete framework for energy measurement, including instructions and downloads for hardware, software, and instrumentation (both physical instrumentation for the device to be measured and software instrumentation for the workloads being executed). LEAP can measure multiple channels in parallel at sample rates from the low kilohertz through 10kHz and beyond (up to a maximum of 80kHz depending on the hardware used and the number of channels being sampled). This allows energy data to be acquired for multiple components simultaneously. Most importantly, the LEAP software instrumentation and synchronization mechanism allow

[35] https://www.wattsupmeters.com/secure/index.php
[36] http://lasr.cs.ucla.edu/deep/

samples obtained during execution of a workload to be matched precisely to the software executed during the sample period.

The basic idea behind LEAP is that a multichannel sampler can simultaneously acquire energy consumption data for multiple independent components by measuring the voltage drop across a "sense resistor" attached to the power source for each component. Acquisition devices of this nature can be external (i.e. USB) or internal PCI devices, making them fast and portable. Making use of this data requires the samples to be synchronized to each other, which happens by virtue of the multi-channel sampling process. It also requires the samples to be synchronized with the specific parts of the workload being measured – something that does not happen automatically.

To synchronize the software executing on the system with the samples, the system generates a sync signal which is sampled alongside the data. This is done by periodically toggling the value of a pin on a serial or parallel port up and down whenever data is being acquired. This periodic signal is measured alongside the other components. When the kernel toggles this signal, it reads and records the value of the nanosecond granularity Time Stamp Counter (TSC) on the CPU using special inline assembly calls that complete on the order of a few operations. After data has been acquired, careful analysis of the sample data and the TSC log created by the kernel enables interpolation of TSC values for the samples that were acquired between toggle events.

In order to identify the samples acquired during the execution of the workload, a small amount of instrumentation must be inserted into the software.[37] This instrumentation, called "energy calipers," reads the start and end times of the measured event using the TSC, either with

[37] Other instrumentation mechanisms using "probes" for user- and kernel-space code exist that do not require recompilation.

inline assembly or by calling the LEAP-specific utility `getticks`, and writes those times to a log file along with some human-readable information naming the event. Since the previous step provided TSC values for every sample acquired, the final step is to identify the samples taken between the start and end times and perform analysis on those samples.

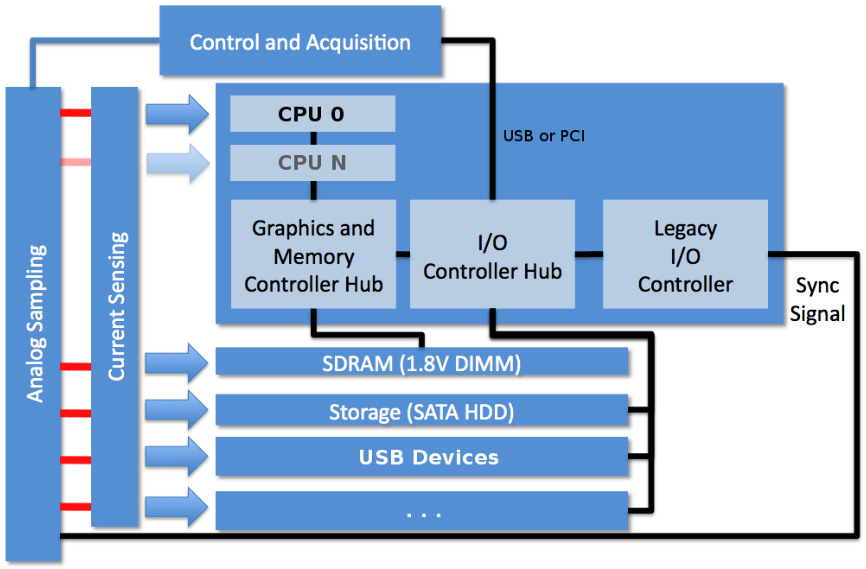


Figure 3. The LEAP architecture diagram [84].

Figure 3 shows a general schematic for the LEAP architecture. The “Analog Sampling” component is the multichannel sampler, which reads the voltage drops across the current sensing resistors attached to the power source for each component. Note that “component” can include the CPU or CPUs, memory (per-DIMM), storage, USB devices, or any other component for which the power source can be isolated and instrumented. Software (“Control and Acquisition”) drive both the sampler and the “Sync Signal” generated by a legacy component (a serial or parallel port).

LEAP overhead is generally quite small. The traditional approach using offline data analysis as described uses only about 1% more energy than the same workload without LEAP (not counting data analysis). A recent development is Online LEAP [84] where energy measurement information is produced in “real time.” However, since sample analysis must occur during execution, this approach requires additional energy – approximately 5% more than the same workload without LEAP. Because Comptool is an offline tool and does not require data in real time, it uses LEAP in an offline fashion.

Integrating LEAP instrumentation into Comptool was straightforward. Doing this primarily required the addition of LEAP "energy caliper" instrumentation around its "compress and send" functions. The LEAP instrumentation could then record the start and end times of each Event, logging the times along with a name describing the event being measured. Thus, every time Comptool performs a particular compression Event, the energy is measured as it happens – just like other statistics such as runtime, compressed size, compression ratio, and so on.

However, in the offline LEAP paradigm, every step in the sampling and analysis of LEAP data is performed manually. That is, the sampler is typically started as a separate process, the workload is executed, sampler is stopped and the data is analyzed to extract the energy data – all manually. This process (start sampler, run task, stop sampler, analyze data) assumes that a single experiment can be encompassed by a single run of the LEAP sampler. Unfortunately, many runs of Comptool cannot be run as a single LEAP experiment in a practical way, because high-frequency sampling (e.g., multiple floating-point values generated at 10 or 20kHz) generates a huge amount of data which raises the cost of analyzing the data. Instead, it was obvious that Comptool would be more robust and user-friendly if it could manage its own offline LEAP experimentation.

Thus, Comptool needed to be LEAP-aware – that is, it needed mechanisms to control LEAP sampling and data analysis from within the application, rather than expecting it to be managed by a human being. To do this, I created a Python module that instantiates a LEAP control object capable of starting and stopping experiments, performing analysis, and returning results to a Python application. This module was designed to be useful for any Python application that would benefit from integrated LEAP sampling. Additionally, because of the time cost of analyzing the large volume of experimental data generated by Comptool, I wrote an analysis tool in C which extracts the energy data many times more quickly than the original Python tools.

LEAP instrumentation code must be able to read the TSC to mark the start and end of workload components. This is trivial to read in a C program using inline assembly. However, inline assembly isn't generally available to interpreted languages such as Python. In the process of figuring out how to add and improve upon the LEAP instrumentation in Comptool, I encountered some interesting issues. These issues, which are described in the next section, led to significant improvements to Comptool and an additional contribution to LEAP.

5.3.2 Timing Mechanisms

Accurately gathering timing information between compression events was a source of error early in Comptool's design. While Python offers many native mechanisms for timing, Comptool needed a timing mechanism that could generate data synchronized with the LEAP instrumentation. In the typical LEAP experiment approach, a call is made to read the CPU's Time Stamp Counter (TSC), a source of nanosecond time data. This call is extremely inexpensive in compiled languages that support inline assembly, such as C. However, for interpreted languages like Python or bash, such an assembly call is not normally available.

To provide LEAP-compatible timing data for previous experiments using interpreted code (such as the bash shell), I created a small, standalone binary utility that uses the same inline assembly used for C-language LEAP workloads. This program, the aforementioned `getticks`, simply reads the TSC and prints the value to standard out. `getticks` is easy to integrate into scripted applications and works well for many experimental workloads.

However, even though `getticks` is an extremely simple application, its process overhead was simply too costly for many of the short tasks that Comptool needs to time, because of the same issues faced by compression utilities (described in the previous section). In short, starting a new process costs as much or more than the entire estimated send time for small amounts of data at high throughputs. For example, sending 64K uncompressed at 100mbit/s should take 0.005s (ideally). However, on the test systems used, the overhead of starting another process is approximately 0.003s. This meant that calling getticks before and after each compression event was actually inflating the test time unrealistically. Put another way, the process overhead was limiting the minimum size of events that Comptool could accurately measure.

To solve this problem, I created a Python module that allows Python programs to read the TSC as a native function, rather than calling an external process to do so. This native "Python TSC" module virtually eliminated the cost of reading the TSC. However, when this overhead was eliminated, the error caused by bandwidth simulation mechanisms became apparent.

5.3.3 Simulating Bandwidth Limits

Bandwidth is critical in determining the best compression choice because in low-bandwidth situations it is often the dominant communication bottleneck. If Comptool should be able to simulate many scenarios, it is important for it to have a flexible and reasonably accurate way to simulate bandwidth limitations. Comptool's bandwidth limiting mechanism needs to be reasonably accurate in order to emulate a variety of network bandwidths. However, it is not critical for the bandwidth limiting to be absolutely perfect because Comptool is meant to compare significantly different throughputs. It is acceptable if it can reasonably emulate various types of networks with broadly different bandwidths, such as gigabit and 100-megabit networks, cable modems, cellular phones, and so on.

Since Comptool is a standalone tool, it must rely on a local communication method that doesn't require a remote host. Comptool must also be able to gather timing data in an unobtrusive way while performing this throttling. Comptool is also meant to provide real-world results, not simply simulated results, so it was important that the limitation actually behave as realistically as possible. Finally, since Comptool uses file input and output, it would be preferable for the mechanism to also use that interface.

I initially thought this would be straightforward. However, due to the discrete and fine-grained nature of Comptool's testing approach, in conjunction with a wide range of network speeds and input sizes, finding a workable solution that was also reasonably accurate presented some interesting challenges. The techniques I investigated, as well as their benefits and drawbacks, are outlined below.

5.3.3.1 UNIX pipes, `pv` and `cstream`

Since Comptool is written in Python (for ease of development, clarity and portability), originally used file utilities for compression, and would benefit from threading and parallelism (separating compression and transmission phases into separate processes as done in many AC systems), an obvious choice for bandwidth throttling was to use standard UNIX pipes to manage input and output flow control. While typical pipes and associated programs such as `cat` do not throttle data in a configurable way, a number of utilities are designed to do just that. The utility `pv`, included in Ubuntu Linux, is akin to a version of `cat` with numerous bells and whistles, including throttling.

Using pv's limiting feature to control the output rate was my first choice. While the details are somewhat more complicated, the basic event unit of a Comptool experiment is a timed compress-and-send operation on some chunk of input data. When using file utilities, the event is functionally similar to the following bash code:

```
$ time -o /tmp/time.txt gzip input | pv -L $LIMIT > output
$ calculate_stats /tmp/time.txt output
```

Figure 4. Pseudocode for basic Comptool event.

Here the entire compress-and-send operation is timed, with pv ensuring that the output rate will not exceed $LIMIT bytes per second. Following compression, the size of the compressed file output is measured, and is recorded along with the relevant timing information captured by the utility time and stored in /tmp/time.txt for analysis. (Again, for Comptool many internals are slightly different, for example the timing is all captured within Comptool as described above).

After much experimentation, I found better accuracy using the pipe limiter cstream as opposed to pv. cstream [96] specifically advertises accuracy, including more options for limiting both the bandwidth and the buffers used. In particular, cstream can treat a limit as something that is to be enforced either over the long run over many read/write calls, or something that is to be strictly enforced at every call – a cap never to be exceeded rather than an average limit to be maintained.

This method worked properly when the input consisted of large files, such as portions of a large file or artificial test data. However, it was very inaccurate when small chunks of files were measured while being compressed and sent with high limits (emulating fast networks). In these cases, the runtime of the tasks was far too long, making Comptool useless for testing short duration events. This problem occurred for the same reason that getticks and compression utilities were problematic for timing, namely, process overhead. The overhead of the pipeline design overtook the time of the workload itself, rendering the result useless.

Confusingly, at other times, writes seemed to take no time at all, even though they should have been much larger than the process cost. This happened because both `cstream` and `pv` buffer input internally. Naturally, both the buffer and the bandwidth limit counters were reset each time `cstream` was invoked (at the start of each compression event). As a result, each new `cstream` process would have an empty buffer ready for data. This meant that if an event was longer than the process startup cost, but small enough that the write would fit into the buffer, it would complete unrealistically fast – the buffer would consume it much faster than it should have.

5.3.3.2 A Persistent FIFO

To avoid both the startup costs of the limiting process and the buffering inaccuracy just described, I modified Comptool to use a UNIX named pipe, or FIFO. A FIFO (first in, first out) is a UNIX pipe with persistence beyond the life of any process and with an addressable name in the file system. In this design, Comptool connected a FIFO to a single `cstream` instance that ran for the duration of many individual tests. In this way, the buffer would be as full as the workload could manage, and the state would persist across multiple compression events. This allowed the emulated bandwidth limiter to demonstrate "sustained behavior" more frequently.

Putting it all together, Comptool used `cstream` and the FIFO in the following way. First, it did the following:

```
$ mkfifo ctpipe
$ cstream -t 13107200 -b 4096 -B 4096 ctpipe -o - &
$ cat > ctpipe
```

This created the FIFO named `ctpipe`. Next, it started a cstream process reading from the FIFO `ctpipe` with a 100-megabit limit and 4K buffers (imposed by `cstream`). The `cstream` instance was backgrounded, as indicated by the ampersand. Then, an instance of `cat` opened the FIFO for writing but did not provide any input. This is a well-known hack to keep a FIFO open (otherwise it would close after the first Comptool Event sent an EOF). Another feature of `cstream` is that it can be configured to simply discard data ("-o -") rather than redirecting it to `/dev/null` which requires kernel intervention.

With this setup, the FIFO `ctpipe` can be written to or read from by applications as though it were a device while providing bandwidth limitation continuity – all for the cost of a single process startup that could be performed before any timing began. Compression methods wrote to the FIFO, while the kernel handled moving data to the `cstream` instance in the background. By making the `cstream` process high-priority (not shown), I ensured that delays due to scheduling were minimized. Unfortunately, while this architecture eliminated the startup overhead associated with the pipe limiter, fixing this problem only made more obvious the fact that many writes were completing too quickly.

5.3.3.3 Kernel Buffering and Bandwidth Calculations

After much investigation, I realized that, in addition to the buffering performed by pipe-using applications, the kernel itself buffers communication between applications using pipes and naturally, this process-kernel communication is not throttled. In other words, even if the `cstream` FIFO was throttling the data *it processed*, the kernel was not throttling the write from Comptool to the FIFO as it passed through the kernel (necessary for IPC). As a result, writes that were smaller than the kernel buffer size (typically 64K as of this writing[38]) would often complete too quickly and thus could not be properly throttled with a pipe-based architecture. While the size of this buffer is adjustable in newer kernels, it is not adjustable for the kernel on the LEAP test system.

This issue could be partially addressed by performing more repetitions of each test and averaging them, although this would extend already lengthy test times. Another partial solution would be to use much larger pieces of input. However, because I wanted to keep data at realistic sizes, I did not feel like it would be accurate to simply concatenate test data together to overcome this issue.

A compromise mitigation strategy I tried was to "spam" the kernel buffer with junk data before performing the timed operation on realistic input sizes, essentially ensuring that the kernel buffer was always full. However, not only did this also extend experiment times (since the junk data still had to be sent over the throttled pipe), but if a compression method was slow and the bandwidth was high, the junk data could drain from the buffer fast enough to make enough space in the kernel buffer for the compressed data – again resulting in a write that was unrealistically fast.

Ultimately, this became an especially problematic issue for Comptool, because it identifies the "best" choice from many different operations. When some event is unrealistically fast because of buffering effects, Comptool ends up making recommendations that were exactly the opposite – methods appearing fast in the results because they took just long enough for the kernel buffer to drain in time to consume the input unrealistically fast.

[38] http://stackoverflow.com/questions/4624071/pipe-buffer-size-is-4k-or-64k

5.3.3.4 TCP Sockets

Given all this trouble with the pipe approach, an obvious question is “why not actually perform network communication?” In fact, I also tried the approach of using a TCP connection to localhost, in order to retain the “standalone” design of Comptool. This option used TCP networking, but treated the socket as a file (a feature of Python) so that Comptool could use compression utilities on it directly. This method worked similarly to the `cstream` method (with a persistent background listener), but used the Linux kernel's traffic shaping tools to limit the bandwidth similar to an early design for `drcp` bandwidth limiting (for Datacomp testing).

The intended benefit of this approach was that it would provide persistent limiting of the bandwidth, would rely on the kernel for TCP transmission and throttling, and would limit network transmission with the same method I intended to use for `drcp`. However, the overhead of the network write calls was too great, resulting in unrealistically *long* transmission times, in the hundredths of a second, rather than the thousandths of a second required to correctly simulate 100-megabit traffic.

5.3.3.5 Bandwidth Calculation Method

Ultimately, in order to accurately simulate the delays and timing for all sizes of input, I changed to a hybrid file- and timing-based approach. This approach used actual file writes to ensure that writes paid all the usual system call costs, along with a timing component to ensure that writes could not complete too quickly. The current mechanism calculates the shortest possible time to complete the given write based on the bandwidth limit and output size. Then it performs the write while timing it. If the actual write operation completes *faster* than the calculated time dictates, Comptool simply waits out the remaining time before returning.

Comptool “spins” while it waits, counting the number of loops spent waiting. While this is a source of CPU and energy overhead, the alternative was less accurate. Originally, Comptool would sleep the remaining time using the Python call `time.sleep()` (which supports the use of high granularity floating-point time values). However, the Python `time.sleep()` function may yield the processor, requesting to be awoken after the remaining time has elapsed. Put another way, `time.sleep(seconds)` guarantees only that it will sleep *at least* the specified number of seconds. This led to longer than expected delays, because Comptool was forfeiting its time slice, delaying it (at least) until it was next scheduled. Using the hybrid approach with “busy waiting” as described has enabled the best accuracy to date.

5.3.4 Contention Workloads

Since my research is on Adaptive Compression in uncontrolled, general-purpose environments, it is important to understand how other processes contending for system resources might affect compression strategy selection. Comptool includes support for launching "contention workloads" – applications meant to consume system resources (primarily CPU time). Comptool supports this by allowing the user to specify an arbitrary command to run during the test workload. This command would typically start one or more programs to run in the background. Following the tests, Comptool kills the background processes.

For my purposes, rather than attempting to mimic “realistic workloads”, a task fraught with pitfalls and inviting real-world contention, I instead used “worst case” workloads that monopolize computational resources in a predictable way. My rationale was that if Comptool was able to show benefit even during times of intense load, it would then be reasonable to conclude that it could provide benefits during more modest or intermittent contention. My standard CPU contending workload simply uses bzip2 to compress pseudorandom data generated by the kernel (`/dev/urandom`), discarding the results by writing to `/dev/null`. In order to create multiple levels of contention, my scripts can run multiple bzip2 processes, consuming approximately *1/c* CPU resources, where *c* is the number of cores on the system.

5.4 Using Comptool

5.4.1 Basic Mechanism

Comptool takes in a list of files, which represent a sequence of data being transferred over an IO channel, such as writing to a disk or transmitting over a network. Processing each file one by one, Comptool breaks it into chunks, compresses each chunk of the file with a single algorithm, and writes the performance values of the compression operations into a database before proceeding with the next compressor.

"Chunks" are used to mimic how an adaptive compression mechanism like Datacomp might quantize the file to provide opportunities to adapt the compression strategy. Currently the chunk sizes used are 32k, 64k, 128k, 256k and 512k. I settled on those sizes for several reasons. First, previous research [94] [49]) indicates that compressing in chunks smaller than 32k is often suboptimal. Including a wide range of chunk sizes also provides variety and some assurance that Comptool can capture meaningful differences between sizes. I have also observed compression ratio benefits for algorithms such as DEFLATE to level off around 256k. However, since some algorithms (such as xz) benefit greatly from much larger inputs, and because adaptation imposes overhead for each chunk (increasing the value for larger chunks), I also provide 512k chunks. Finally, as a baseline, each file is also compressed and sent as a single unit regardless of size.

Comptool's basic algorithm is shown in Figure 5.

```
start_contention_workload
start_bandwidth_limiter
for file in file_list:
      for chunksize in size_list:
            chunk_list = split_file_by_size(chunksize)
            for compressor in compressor_list:
                  for chunk in chunk_list:
                          start = time()
                          results[chunk] = compressor.time_comp_send(chunk)
                          end = time()
                  record = calculate_results(start, end, results[chunk])
                  store(record)
stop_contention_workload
stop_bandwidth_limiter
```

Figure 5. Pseudocode for Comptool.

Originally, files were simply split at whole values of the given chunk size. For example, if the input was 128K and the two chunk sizes used were 32K and 64K, the 32K chunks would start at bytes 0, 32K, 64K, and 96K while the 64K chunks would start at 0 and 64K. However, this ignores the possibility that the range of bytes between 32K and 96K would be best compressed as one 64K chunk (with 32K chunks at either end).

Because this is exactly the kind of situation we want Comptool to be able to identify, chunks are now started at the interval of the smallest currently configured chunk size such that the resulting chunk still fits within the file. Thus, while the starting positions for 32K chunks would be unchanged (0, 32K, 64K, and 96K), in this scheme there would be an additional 64K chunk starting at 32K (with the full list being 0, 32K, 64K). There would be no 64K chunk starting at 96K because that chunk overlaps the end of the file. This means that many tested chunks include overlapping sequences of data.

After all operations for all files have completed, a greedy operation is used to search through the result set. This search identifies the sequence of chunks with the lowest resource consumption (for the given goal) that fully “covers” the file list (this process is described in

Section 5.4.2.2). Because many bytes in tested chunks overlap, the offline analysis code must ensure that each byte of input is only used *once* in the selected output strategy. This is straightforward since each chunk includes its original starting offset and its length.

Finally, while the chunk sizes are not configurable at the command line, they are (like the set of compression methods) defined as a simple Python list:

```
SIZE_LIST = [0, 32768, 65536, 131072, 262144, 524288]
```

As such, the chunk sizes are trivially adjustable[39]. Additionally, the mechanism used to split files into discrete pieces for analysis is abstracted into its own module. Thus, if an entirely different "chunking method" is desired, it can easily be replaced.

As a simple usage example, consider a single 260K file, which could be a standalone file or part of a larger sequence of files. With the chunk sizes 32K, 64K, 128K, 256K and 512K, a 260K file would be broken into 25 chunks:

- one 260K "chunk" – the complete file – used for testing compressors on whole files
- zero 512K chunks (the file is too small for this to be meaningful)
- one 256K chunk starting at byte 0 and one 4K chunk after that (which, for the purposes of record keeping, is counted as a 32K chunk, since 32K is the smallest size)
- five 128K chunks (starting at 0, 32K, 64K, 96K and 128K), and one 4K chunk
- seven 64K chunks (starting at 0, 32K, 64K, 96K, 128K, 160K and 192K) and one 4K chunk
- eight 32k chunks (starting every 32K) and one 4K chunk

For each chunk (the Data) and Method, compression and transmission is performed in the user-specified bandwidth and process contention Environment, with relevant statistics logged to a

[39] The special size "0" indicates "full file compression" (no chunking).

database. After all chunks have been processed and their results stored, the best strategy is extracted using a greedy algorithm.

5.4.2 Finding the Best Strategy

5.4.2.1 Overview

Given a set of Events and Results generated by Comptool, the best Strategy is one that minimizes the cost with respect to some measured property, while covering every byte in the input once. This strategy can be described as the best "sequence of choices" selected from all the existing possible choices. In this way, finding the best strategy is a kind of "Knapsack Problem." [97] Due to the complexity of finding the truly "ideal" Strategy for a Scenario (as discussed in Section 5.5.1), Comptool uses a greedy approximation when analyzing Events.

5.4.2.2 The Greedy Best Algorithm

The "Greedy Best" strategy is the simple mechanism used by Comptool to identify the best strategy from a set of Events generated by Comptool. Given a set of Events generated in the processing of a file, and some property of interest, such as the compression ratio, the greedy best algorithm looks at all Events (of all lengths and methods), starting at the beginning of the file (offset 0), and selects the one with the best Result statistic of interest. Next, it advances the offset based on the uncompressed length of the "winning" Event (e.g., 32K if the best chunk was 32K uncompressed). Finally, it repeats the process until the end of the file is reached. This generates a "greedy strategy" – a sequence of events approximating the maximum performance for the given property. The greedy best algorithm is described in pseudocode in Figure 6.

```
offset = 0
winners = []
while offset < length(file):
      best_chunk = get_best_event_chunk_for_value(file, statistic)
      winners.append(best_chunk)
      offset = offset + best_chunk.uncompressed_size
```

Figure 6. The Greedy Best algorithm for Comptool.

5.4.2.3 Example – Compression Ratio

Consider the **partial** table of chunk data for a 128K file as shown in Table 11. In this sample, there are three chunks that start at offset 0 in the test file (Events 1-3 in the table). Events 1 and 2 are 64K and were compressed using Method 0 and 1, respectively, while Event 3 is 128K and was compressed using Method 1. Event 4 is 64K and was compressed with Method 0.

Event	Offset (K)	Algorithm	Chunk Size (K)	Time (s)	EOR (K)	Compression Ratio
1	0	0	64	.1	640	.25
2	0	1	64	.07	914	.28
3	0	1	128	.13	984	.26
4	64	0	64	.08	800	.27

Table 11. An example of Comptool experimental data. (All rows are not shown.)

If we are trying to optimize for size, we would be interested in the compression ratio achieved for each Event. Looking at the Events starting at offset 0, the best compression ratio is 0.25, for Event 1. Event 1's uncompressed size is 64K, so advancing the offset to 64K, there is only one Event to choose, Event 4. So our greedy best Strategy for compression ratio is Event 1 followed by Event 4.

5.4.2.4 Example – Effective Output Rate and Bytes per Joule

Suppose we are trying to find the Strategy that leads to the shortest overall time. If we consider run time for each Event, smaller chunks would generally appear to be faster than longer inputs by virtue of length. Thus, considering *time* alone, we would choose Event 2 with its run time of 0.07s, followed by Event 4 (which starts at offset 64K) with its runtime of 0.08, for a total runtime of 0.15s.

But the time in seconds for an individual Event is not actually the value in which we are most interested. We are most interested in increasing the rate of transmission across the entire Scenario since we want to minimize the total runtime. So, instead of looking at the run time of a particular choice, we need to describe the run time of the Event in proportion to its uncompressed size in order to take input length, the efficacy of compression, and the time used into consideration.

Thus, rather than considering time alone, we consider the Effective Output Rate (EOR), which is calculated as:

$$EOR = uncompressed\_size / operation\_time$$

… where the operation time in this case includes compression and transmission (an important distinction which will be discussed later). Note that the EOR includes the operation time, which includes both time spent in compression and in transmission, two values that may overlap in AC systems depending on the architecture. Note also that the EOR is based on the *uncompressed size* – not the compressed size – since the compressed communication operation is actually *communicating* the number of bytes in the uncompressed payload. If we looked at the output rate in terms of compressed output size for multiple compressors with the same compression time, the more effective compressors would appear to have *worse* performance because they send fewer bytes (shown in Table 12).

Compression Ratio	Compression Time	Compressed Output Rate
0.25	1.0	0.25
0.50	1.0	0.50

Table 12. Compressed output rate can penalize more effective compressors.

By looking at EOR, we can compare large and small chunks in a more equitable fashion. Armed with this equation, we see that the best EOR for a chunk starting at offset 0 is 984K/s, observed during the processing of chunk 3 which used Method 1 on the entire 128K file. With a Strategy based on time alone, we would have chosen Event 2 followed by Event 4, for a combined runtime of 0.15s or an effective output rate of 853.3K/s.

The procedure for optimizing energy is similar to that used to identify the best strategy for time. We can treat energy Results acquired by LEAP like time in the sense that we can calculate a Bytes per Joule (BPJ) value instead of EOR and scan the results in a similarly greedy fashion. The end result is a strategy that describes the best greedy sequence of events from the perspective of energy minimization.

5.4.2.5 Breaking Ties and Other Details

If priorities are kept discrete and non-overlapping (i.e., no two resources can have equal priority), it is fairly straightforward to "do the right thing" for a secondary priority without any additional instruction. For example, if compression ratio is the statistic of interest, Comptool will choose the event with the best compression ratio, breaking ties for CR by choosing the event with the greatest EOR or BPJ. By the same token, if when optimizing for throughput or energy two events tie for best EOR or BPJ, it is costs nothing to choose the event with the best CR.

In fact, this is how the greedy algorithms in Comptool (and Datacomp) function. If you choose to optimize for space, you get time/energy as a secondary priority automatically. If you choose time/energy as your first priority, you get space as an automatic secondary priority. Additionally, both Comptool and Datacomp break CR ties by choosing the largest size input chunk to maximize the benefit over more bytes. Always making this tie-breaking choice is possible because the other mappings – choosing the best time but the worst CR or the best CR and the worst time – do not map to outcomes that users are interested in. As a result, priorities for Comptool are simple – find the best strategy for time, energy, or space.

5.4.3 Experimenting with Comptool

Comptool can be used in experiments in a number of ways. First, Comptool can be used to identify the best Strategy for compressing a single input under a set of conditions. Intuitively, running Comptool on input once should provide that answer; the list of Methods used can be extracted from the Events. However, since each discrete task consists of one Event (one execution of a Method on a Data chunk), obtaining values with statistical confidence requires multiple repetitions of the same test (or multiple repetitions of similar tests) when optimizing for time or energy. This is because while compression ratio will not change on repeated tests, run time can change based on transient environmental factors even in controlled experimental conditions.[40]

The second way that Comptool can be used, and the most common way it is used in this work, is to identify the best average performance and choices for given classes of data. Testing a single file in isolation is interesting, but of limited value unless you are certain that your workload will only consist of that type of data. Generally, we are more interested in the best average strategies to use for general classes of data and the performance differences across data types, rather than the single best strategy to use for some specific input.

To meet that need, Comptool results can be analyzed in an aggregate fashion. By identifying the best strategies over a series of similar scenarios and aggregating the results, we can easily identify how well the average best strategy performs as compared to any static strategy, or any other dynamic strategy we choose to construct from the captured results. For

[40] Comptool actually performs two executions and averages them together to improve accuracy. However, it is still important to perform multiple repetitions for statistical confidence.

example, given *n* pieces of similar input (say, different Wikipedia pages), we can use Comptool to determine the best strategy for each and then compare the average performance of the "best strategy" against the static performance of gzip or null.

Digging deeper, we can analyze the commonalities of choices for similar sequences (e.g., 25 different samples of Wikipedia data) and compare the differences between strategies for different data types (e.g., comparing the best strategies for YouTube versus Wikipedia). For example, we might identify that above some threshold, no compression algorithm was able to improve throughput, or that under a set of conditions, LZO was far and away the most selected strategy (or that xz was never selected). We might also be able to identify patterns in the choices, such as that a particular algorithm regularly appears as the best choice under low throughput conditions, but not otherwise.

Comptool keeps many statistics for all Events, including the file extension when available, the compression time and send time, and of course all Method-related information including algorithm strength, chunk size and so on. This allows Comptool Results to be mined for other interesting relationships and results regarding a long series of experiments.

Finally, Comptool can also be used as a test harness for compression algorithms. Internally, Comptool represents compression choices as a data structure containing the algorithm and all its parameters. From that data structure, Comptool generates every unique combination of those parameters. Thus, Comptool's output provides a means to compare the effect of various parameters. This can be done within a particular command (e.g., comparing the effect of "strength 1" versus "strength 2"), or between commands (e.g., comparing the difference of strengths 1 and 9 for algorithm X versus algorithm Y).

5.5 Limitations

Comptool has various limitations, some of which are a result of its standalone and scripting-language design, while some are unavoidable given the characteristics of modern computers. Some of these limitations were described above when discussing Comptool's design. Fundamentally however, Comptool's limitations illustrate a key challenge for designing general, multi-purpose AC systems: because performance in these systems is critical (even for testing tools like Comptool) and the "profit margins" in challenging scenarios can be small, design choices in one area of the system can easily impact the properties of the system in unexpected ways.

For example, it is not practical to test all potential chunk sizes. Comptool's maximum chunk size is 512K; it cannot speak to the benefits of using one-megabyte chunks. However, as we will see in the discussion of dzip results (Section 8), this surprisingly does affect the maximum compression achievable by the tool for some methods. Additionally, while the use of some heuristic (such as a greedy algorithm) is acceptable given the problem space, such algorithms result in suboptimal solutions. As a result, while Comptool is designed to be a "compression oracle" that can identify the *best* choice for a given scenario, its results should more literally be thought of as a very good estimate for the best approach. I discuss some of these limitations in more detail in the following sections.

5.5.1 Time Cost of Brute Force Search

Probably the most obvious limitation of Comptool for most users is that individual test runs can take an enormous amount of time depending on the specific scenario. While Comptool is not a "live" Adaptive Compression system dependent on "real time" results, this cost can still be daunting.

For example, consider a test using one megabyte of relatively uncompressible data, being compressed and sent over a 64KB/s link. One megabyte requires 16 seconds to transmit uncompressed on such a link. Each experimental Method (created by the permutation of unique methods) requires an additional transmission operation. For example, compressing and sending the entire input with “null”, LZO, gzip, bzip2, and xz each cost around 16 seconds (ignoring compression cost and assuming the data is not compressible). Additionally, gzip and xz have multiple strengths to test, the tests can be run at different CPU frequencies or with different contention workloads, and of course breaking down the file into multiple chunks of various sizes results in compressing each input file multiple times with each discrete method (the 260K input from Section 5.4.1 was broken into 25 individual chunks). LEAP energy measurement also adds overhead due to starting and stopping the sampler, in addition to processing the experimental data.

The end result is that Comptool runs can take a very long time, especially if many types of data and experimental factors are being analyzed. As a concrete example, in my tests, testing a one-megabyte sample of one type of data (in one environment) at 100mbit/s (with no repetitions for confidence) takes approximately seven minutes. The same test at 1Mbit/s will take considerably longer (but not 100 times longer because some costs are not dependent on bandwidth). Of course, these elements are all tunable – I have tested many factors for this work that others users of Comptool may not find interesting. In contrast, using Comptool to identify the best gzip strength level or to compare gzip and LZO results for a single input without bandwidth restrictions would not take long at all. However, it is inevitable that using this approach with more than a trivial set of options will take a considerable amount of time.

5.5.2 The Cost of a Scripting Language

While there is an unavoidable time cost to performing a "full factorial" test on a large number of factors, one set of limitations is based on the design choice of using a scripting language for Comptool. While programs in interpreted languages are generally much easier to understand, develop and modify, using an interpreted scripting language such as Python is simply much more computationally costly than a compiled language, because these interpreters take considerably more time to complete a statement than compiled code running directly "on the metal."

Comptool's basic operation is not especially time sensitive, and since I intended compression to be performed by compiled utilities, I thought that the benefits of interpreted code outweighed the costs of writing Comptool in C. However, the overhead of using interpreted languages, and in particular the use of command line utilities, imposed some hard limits to throughput based on the processing power of the system. This, in turn, limits the resolution of Comptool because it limits environments that Comptool can accurately emulate.

For example, suppose that, using various workarounds, Comptool's compress-and-send cycle for some algorithm could be reduced to 0.0001s. In an ideal gigabit network capable of sending 134,217,728 bytes per second, 13.4K of data would be sent during that one-thousandth of a second. If this is the minimum time resolution Comptool can measure or perform, then it is not reliable for any payload less than 13.4K in that environment. This may or may not be a significant limitation depending on the needs of the experimenter.

Writing Comptool in a compiled language would have some benefits. In particular, users might have more confidence that the maximum performance was being wrung from the system, especially if they must perform tests on small inputs and fast networks. A compiled version (or a version using a different interpreted language) could also thread the tasks of compression and throttled transmission. Python's threading mechanism is primitive and is limited in counter-intuitive ways by a global interpreter lock (GIL) [98].[41] Additionally, since Comptool compresses data in relatively small chunks, it isn't the case that sending operations are stalled until some "large" amount of input (e.g., 5, 10, or 50M) is fully compressed.

However, writing Comptool in a compiled language would have had significant drawbacks as well. First, the major mechanisms of Comptool involve reading and writing files, cutting input into different sized pieces, timing, and writing logs – tasks well-suited to scripting languages. Writing compiled code to handle these tasks is possible, of course, but would be more complex, more costly in terms of development time, and more difficult to modify. Additionally, although writing Comptool in C would have doubtless improved resolution at high throughput levels, it would still have been subject to many of the same overheads and limitations as the Python version. For example, using a command pipeline would still introduce process overhead and regardless, `read()`/`write()` or `send()`/`recv()` calls would still be subject to system call overhead imposed by the kernel. And of course, the same buffering issues faced by the Python version would be faced by a compiled implementation. As with the Python version, a different architecture could eschew system calls altogether and perform all tasks in memory, but the further the design goes in this direction, the closer it comes to simulation and the further it diverges from actual application behavior.

Another issue is that regardless of the architecture or implementation language, the actual compression work would almost certainly be performed by libraries compiled to machine code (such as zlib) that are ultimately shared by most compression applications regardless of high-

[41] While it is possible that Comptool could be written to avoid issues with the GIL, I did not want to base my projects success on my ability to navigate this thorny and contentious engineering issue.

level language. A given system can only compress data so fast with a given method; if that minimum time can already be measured with an interpreted language, then writing the tool using a compiled language won't improve the resolution any further. Because of this, once I moved to a "pure Python" architecture for timing and compression, the benefits of a compiled architecture seemed small.

5.5.3 The Cost of Being "Standalone"

Designing Comptool to be a "standalone" utility has costs, because certain activities cannot be directly measured. For example, Comptool does not capture the decompression cost or temporal effect of decompression on the system alongside compression. A standalone system cannot, in real time, measure the compression and decompression costs since they would overlap on the same hardware. Instead, by default, Comptool discards the compressed data, in essence showing the best strategy for the compressing side assuming that the decompressor can always decompress faster than the compressor can compress. In practice, this may not be a significant limitation, because decompression is generally faster than compression. However, it is important to recognize that this may not always be the case, especially when there is significant asymmetry between the compressing and decompressing hardware.

That said, Comptool can measure decompression costs, just not at the same time as it measures compression costs. In this mechanism, previously-compressed data is transmitted and decompressed, with only the receipt and decompression performance being measured. Because these additional tests would increase already lengthy experiment times, and because other experiments capture decompression time, I did not evaluate decompression performance using Comptool.

5.5.4 Greedy != Not Optimal

A final limitation of Comptool is due to the greedy search mechanism. While the greedy algorithm is used as an optimization due to the complexity of finding the ideal solution, it is inevitable that the greedy approach does not always result in the best strategy. For example, Table 13 shows a set of values for which the greedy algorithm is not ideal.

Event	Offset (K)	Algorithm	Chunk Size (K)	Time (s)	EOR (K)
0	0	2	32	.03	1067
1	0	0	64	.1	640
2	0	1	64	.07	914
3	0	1	128	.13	984
4	32	1	32	.05	640
5	64	0	64	.08	800

Table 13. The greedy best algorithm is not ideal for this data.

Starting at offset 0 and optimizing for EOR, the best chunk is Event 0, with an EOR of 1067. However, this Event is only 32K bytes long. This forces the next choice to be Event 4 (the only Event in the table covering the 32K-64K byte range), which has an EOR of only 640. Finally, we would choose Event 5 to cover the last 64K. The resulting average EOR would be (1067 + 640 + 800) / 3 = 835.7K/s. However, if we had just chosen Event 3, which covers the entire input from 0-128K, we would have achieved an EOR of 984K/s. Ultimately, I leave the goal of identifying better non-greedy strategies based on Comptool data as a task for future work.

5.6 Lessons from Comptool

Though it was initially disappointing that Comptool cannot perfectly capture the full range of bandwidths, in practice this is perfectly acceptable. Every program is limited by its hardware and software, and every measurement tool has its limits – Comptool is no different. Not only is Comptool's accuracy sufficient for most uses, but the issue with small files on fast networks mirrors a problem relating to high-speed networks in general: small payloads in high-speed networks reduce throughput because the fixed cost of each send operation often bottlenecks the performance. In this sense, Comptool is simply demonstrating an issue faced by other applications that send small files individually.[42]

Ultimately, when one considers the issues arising from compressor choices and opportunity costs (both at the local and distributed levels), it becomes apparent that the ultimate costs and benefits for any compression scheme (adaptive or otherwise) will depend heavily on specific implementation details, especially when operating on high-speed networks. In other words, Comptool can provide informed advice about the best compression choices by using brute force search, and those results are accurate insofar as Comptool is functionally similar to the system to which it is being compared. In the end, Comptool can be simple and generic, or highly optimized for a particular purpose, but not both.

[42] I also considered having Comptool compress and send files in concatenated chunks, but opted against it because that is not how most applications behave.

6 DATACOMP

6.1 Introduction

Datacomp is a locally independent Adaptive Compression system for use in the general computing environment. Datacomp makes all decisions locally without support from any external systems, and relies on a minimum of hard-coded design choices. It makes choices at a granularity suitable for real-world workloads and takes pains to capture as much improvement as possible while also avoiding any losses. It tests a wide range of strategies, monitors several environmental parameters, is able to estimate the compressibility of data, and requires a minimum of modification for use in a wide variety of user-space applications.

At a high level, Datacomp can be described as a wrapper that provides the special functions `dcread()` and `dcwrite()`. These functions are adaptively-compressing versions of the standard system calls `read()`, `send()`, `write()` and `recv()`, and can be used to adaptively improve efficiency using compression for either time or space.[43] Inside a call to `dcwrite()`, Datacomp performs compression, which may be parallelized, and encapsulates the output in a format indicating how it may be decompressed. Datacomp then makes the underlying `send()` or `write()` calls on behalf of the application. An analogous process happens for decompression, where the compressed data is decompressed automatically (possibly in parallel) and the raw data returned to the user. Since Datacomp hides the complexity from the application and is compatible with the most basic behavior of the traditional I/O calls,[44] Datacomp should not require significant changes to the underlying application.

[43] `dcwrite()` and `dcread()` can write and read from both files and sockets.
[44] Basic limitations in this Datacomp prototype are described later in this section, however many existing limitations can be removed with additional software engineering.

Consider a simple file compressor, shown in Figure 7. The program opens two file descriptors, one for raw input and one that will be used to write compressed output. Then, the program reads the data from the input, applies a compression function, and writes the compressed result to the output (using the system calls `read()` and `write()`). Decompression works similarly, but in reverse; the compressed data would be `read()`, decompressed, and then sent to the output using `write()`. A compressed network transfer tool would work in a similar fashion, except that a TCP socket would be created for the output (rather than opening a regular file). The tool would `read()` the raw input from disk, apply compression, and `send()` the output. The receiver would `recv()` the input, decompress it and `write()` the output to disk. In both cases, the compression mechanism must be manually selected by the developer and integrated into the software.

Developers wishing to use Datacomp in either one of these programs would need to make very few changes, as shown in Figure 8. On the compressing side, developers would create a Datacomp File Descriptor (DCFD) using the function `make_dcfd()` by providing a priority (time or space) and the output file descriptor.[45] Then, instead of the two-step process of explicitly performing compression and sending the result, they would simply use `dcwrite()` with the raw input and the DCFD (instead of the plain file descriptor) instead of `write()` or `send()`. Inside `dcwrite()`, Datacomp would choose a method, perform compression, and dispatch the output using `send()` or `write()`, based on the underlying file descriptor and the resource priority provided when the DCFD was created. The end result is that compression is performed automatically and is selected specifically to optimize the selected resource.

[45] The DCFD is actually a structure containing the original file descriptor and all the state necessary for making adaptive choices. In this way, the DCFD is actually more like a C FILE pointer than a true "file descriptor."

```
input = open(path, read)
output = open(path, write)
while (moredata):
      rawdata = input.read()
      zdata = compress(rawdata)
      output.write(zdata)
```

Figure 7. Pseudocode for compressing input manually.

```
input = open(path, read)
output = open(path, write)
dcfd = make_dcfd(output, "time")
while (moredata):
      rawdata = input.read()
      output.dcwrite(rawdata)
```

Figure 8. Adaptively compressing with Datacomp.

Decompression works in reverse; a Datacomp-compressed stream is read using `dcread()` and a DCFD based on the input file descriptor, instead of `read()` or `recv()` on a normal file descriptor. Datacomp acquires the compressed input using the underlying calls `read()` or `recv()`, determine how the data was compressed, decompress it and return the uncompressed output to the application.

The benefit provided by Datacomp to the user is that compression is performed transparently in the service of improving efficiency in terms of the specified resource (which could be user selected). The benefit to the developer is that she does not need to bother with the details of method selection or implementation (which could require adaptation in order to be ideal). The benefit Datacomp provides to researchers is that its highly modular and configurable design makes it a flexible testbed for exploring Adaptive Compression.

While Datacomp hides the details from users and uninterested developers, there is a lot going on "under the hood." As described in Section 3, Adaptive Compression systems are composed of at least four basic components: Mechanisms, Methods, Monitors, and Models. In the rest of this section, I will discuss the design and implementation of these components for

Datacomp, as well as the two example applications I constructed, `dzip` (a file compressor using Datacomp) and `drcp` (a remote copy tool using Datacomp).

6.2 Methods

Datacomp supports the same Methods as Comptool, with two slight exceptions. First, Datacomp uses slightly different LZO-family compressors to provide ostensibly "faster" and "stronger" LZO choices in addition to the default LZO algorithm. Second, Datacomp uses bzip2's "strength level" option (which primarily adjusts the amount of memory used) to provide different "strengths" of bzip2. Datacomp uses the default level of both algorithms as the "medium" strength, while Comptool only uses the default strengths for these two algorithms. In any case, the performance differences provided by these additional alternatives are not large.

Table 14 shows the set of parameters used to identify individual compression Methods. Every parameter of a Method must be explicitly set by Datacomp at compression time (i.e., each Method is "fully determined"). For example, every payload has a "chunk size" even if it is not a perfect match, similar to how the "extra" 4K chunks from Section 5.4.1 were counted as 32K chunks. As a result, each unique Method can thus be described as a tuple of the form *(algorithm family, strength, chunk size, thread level)*. For example, the method corresponding to "zlib, strength -9, compressing 256K in each of 4 threads" can be more precisely and compactly described (using the codes in Table 14) by the tuple *(2, 2, 3, 3)*. "Null" compression is typically performed using the method *(0, 0, 4, 0)*; the "strength level" (second value) is irrelevant for null compression, but the larger chunk size and single thread are most like a typical memory copy.

Value	Family	Strength	Chunk size	Thread level
0	null	Fast	32K	One thread
1	LZO	Medium	64K	Two threads
2	zlib	Strong	128K	Three threads
3	bzip2	NA	256K	Four threads
4	xz	NA	512K	NA

Table 14. Compression method parameters.

The underlying compression libraries used by Datacomp (e.g., libzlib, libbzip2) provide a similar interface, which accepts as input a buffer and length and writes compressed data into an output buffer, returning the compressed length and updating the value of the input length (so that the amount of data consumed is apparent). While the interfaces of these libraries are similar, they are not identical. Furthermore, the setup and teardown process, results, and error codes for each algorithm are not the same.

To handle this diversity, Datacomp uses a unified API for compression and decompression methods. This requires wrapping each compression library with a simple Datacomp "translation layer" which accepts "Datacomp style" parameters and translates them to the lower-level library. These wrappers are extremely simple; the shortest (including includes, comments, whitespace, and both compression and decompression functions) is 73 lines of code, while the longest contains 280 lines of code.

The wrapper determines which method is applied. It accepts as a parameter a structure that contains the input and output buffers, the chosen compressor strength and the length of the input to be compressed (i.e., the chunk size). When compression is called, the wrapper executes the underlying compression function, stores the results in the appropriate buffer, updates the size values, and returns the structure to the code that called the wrapper. In this way, the wrappers handle method execution and return the results in a consistent manner. In addition to simplifying and compartmentalizing Datacomp's code, the use of a translation layer means that it should be straightforward to integrate algorithms in the future, even if they use a very different native API.

Importantly, each wrapper causes the compression operation for each chunk to be "flushed" at the end of the input. "Flushed" compression output stands on its own and is ready to be decompressed; it does not require any other context (e.g., compression or decompression information from any previous or later input).

"Flushing" the compression operation is not necessarily the default for compression libraries. For example, if a compressor is provided a string of 1,000 zeros as input, it may compress the input but not generate any output (if the compressor has not been flushed) since there is (so far) so little output to create. By waiting for more input, the compressor might be able to further improve the compression ratio, such as if the next input consists of more zeros. However, if Datacomp did not flush the compression operation at the end of each input chunk, some compressed chunks might have no output, or might only partially decompress.[46]

Like Comptool, Datacomp attempts to consume data in fixed chunk sizes (if there is enough data). These chunk sizes correspond to Comptool's chunk sizes and are thus the "adaptation quanta" of the AC mechanism in terms of size, with the "time quanta" being determined by the data rate. Every compression operation has a chunk size. If the input available is smaller than the maximum chunk size, then the operation is considered (for record keeping purposes) to be the chunk size of the next largest size.

For example, if a user performs a `dcwrite()` on 8K, the 8K will be sent as a single chunk. Datacomp will consider that chunk to be "32K" for the purposes of record keeping (i.e., it will be recorded as having chunk size "0" according to Table 14). If the user instead sent 40K, the data could, depending on the model and resource priority, be broken up into one 32K chunk and one 8K chunk (which would both be counted as 32K chunks), or as a single 40K chunk (which would be counted as a 64K chunk). In practice, most chunks are one of the standard

[46] Flushing the compressor results in a loss of compression ratio, since it limits the amount of input each operation can compress. However, it is necessary given the possibility of changing compression methods between chunks and the need to limit the amount of data required to decompress in order to acquire some given compressed input. The possibility of using some kind of "shared history" between chunks is an area to be explored in future work.

chunk sizes; Datacomp never explicitly chooses to compress less than 32K. Instead, “short chunks” occur at the end of a discrete piece of input or when the entire input is less than the total size Datacomp has chosen to compress.[47]

The number of threads used for compression and decompression is a property of the Method being used. However, compression threading is not managed by the compression translation wrappers, because this would require the use of compressors that support parallelism (which is currently non-standard) or threading code in each wrapper. Instead, parallelism is a part of the Datacomp components responsible for dispatching compression operations. See Appendix A for more information.

In the next section, I will describe the mechanisms – the machinery – of Datacomp, which include its user space interface, internal behaviors, threading mechanism, and stream format.

6.3 Mechanisms

Mechanisms, as described in Section 3.3.3.4, include the basic approach for integrating AC into a software system as well as the means (e.g., embedded metadata) by which parties using the system can identify the compression types used for the purposes of decompressing. Informally, we might say that the AC mechanism is the structure of the system external to any specific functional modules. This structure determines the abstraction layer where the AC system functions in addition to internal details such as how compression is applied, how the system interacts with user applications, and the “protocol” used by the system to indicate how

[47] I make no claims about the optimality of these choices; rather, they were intended to be fairly arbitrary and make choices easy to model. Determining optimal chunk sizes or means to dynamically adjust chunk sizes are both interesting topics for future work.

data was compressed. Several examples of existing mechanisms in the literature are discussed in Section 4.2.1; this section will discuss the details of Datacomp's mechanism.

6.3.1 User-space Library Interface

Rather than being implemented in a virtual machine, kernel, or middleware, Datacomp is implemented as a user-space library (similar in spirit to AdOC [66]). As a result, legacy applications must be modified in two simple ways to use Datacomp. First, applications must make a function call to create a Datacomp File Descriptor (DCFD) – the "file descriptor" used for Adaptive Compression I/O. Second, applications must use the DCFD along with the Datacomp-specific functions `dcwrite()` and `dcread()` *instead* of the legacy functions `write()/send()` or `read()/recv()`. While the current prototype is not fully compatible with the POSIX API, it is sufficiently functional to implement the test applications described later in this section.

```
make_dcfd(fd, &dcfd, priority)
```

Figure 9. Creation of a Datacomp File Descriptor (DCFD).

To create a DCFD, applications must provide an already open file descriptor or socket in addition to an instruction indicating what resource to maximize (if sending or writing).[48] This is shown in Figure 9. The function `make_dcfd()` is called with the original file descriptor `fd`, a pointer to an empty DCFD structure, and a variable indicating whether to optimize for time, space, or use some static method. Subsequently, the DCFD is used by `dcwrite()` and `dcread()` in the same way that `fd` would be used by the legacy system calls. Finally, while this is not strictly necessary, well-behaved programs will destroy the DCFD using the function call `close_dcfd(&dcfd)` once it is no longer needed.

[48] The actual make_dcfd() function arguments are slightly different (to facilitate testing) but the general effect is the same.

Applications call `dcwrite()` in much the same way that they would make a call to `write()` or `send()`.[49] Figure 10 shows the declaration for `dcwrite()`, modeled directly on `write()` and `send()`. To perform a Datacomp write, the application specifies a DCFD pointer, a pointer to the output data (`msg`), the number of bytes to be written (`len`), and some optional flags.[50] The return value of `dcwrite()` is an integer indicating how many bytes were "written" to DCFD.

```
int dcwrite(struct datacomp_fd *dcfd, void *msg, int len, int flags);
```

Figure 10. The dcwrite() function declaration.

Like the legacy functions `write()` and `send()`, `dcwrite()` *may* dispatch *up to* `len` bytes from the start of the buffer `msg`. If `dcwrite()` were to send `len` bytes, its return value would be equal to `len`. However, `dcwrite()` and the legacy functions can dispatch as few as 0 bytes – returning to the caller having not dispatched any data from `msg`. This is not an error, but rather an indication that no data could currently be written.[51] This can happen if a call is interrupted, if a send buffer is full, if the operation is taking too long, and so on. To cope with this uncertainty inherent in the API, applications using `write()` or `send()` typically make these calls within a loop summing the number of bytes successfully written during each discrete call, leaving the loop once enough data is dispatched.

When a legacy application makes calls to `write()` or `send()`, the function is serviced by the kernel, which consumes the data. Once the kernel accepts the data, it queues it for dispatch at the kernel's discretion. The data may not actually be output until later, although subsequent reads are guaranteed to be consistent.

[49] `dcwrite()` and `dcread()` automatically determine whether the underlying file descriptor is a file or a socket and use the appropriate legacy system call.
[50] The optional flags are currently unused but are accepted for backwards compatibility.
[51] `send()`, `write()`, and `dcwrite()` return -1 upon an error, such as caused by writing to a closed file.

In contrast, Datacomp is a user-space library wrapping these calls – it is actually part of the application itself, bound to the same API used by legacy applications. In order for the application to be stable and bug-free, Datacomp must maintain the same API semantics and guarantees (or a coherent subset of them) as if the application had performed compression and the `write()` itself. In other words, if Datacomp says that *n* bytes were written, then *n* bytes *must* have actually been consumed by the kernel with the same guarantees provided by a call to `write()` or `send()`.

However, if compression is successful (and we hope it is), then fewer bytes will need to be written or sent via the underlying kernel I/O mechanisms. Of course, since Datacomp is making the calls to `send()` or `write()`, the kernel might return a value indicating that not all of the submitted bytes were written. For example, suppose that a user makes a `dcwrite()` call with 256K of input. The application must be able to cope with any amount of this data being written, from zero bytes up to all 256K; if not all the data is sent with the first call, the application can choose to continue sending the rest in subsequent calls. This gives Datacomp latitude in terms of how much of the input to compress at one time. But regardless of the strategy Datacomp chooses, it must be able to actually send or write the necessary output and return a sensible result to the caller.

Suppose that Datacomp takes one eighth of the 256K of input (32K) and compresses it down to 10K. If Datacomp can `send()` the compressed 10K on the caller's behalf, it can return to caller indicating that *32K* worth of the input (the uncompressed size) has been dispatched. To fulfill its end of the bargain, Datacomp must commit to making sure that the kernel has accepted *all* of the compressed data before returning to the application, because there is no sensible return value for the result of sending only *part* of the compressed payload. Thus, Datacomp loops in order to send the entire compressed payload before returning to the caller. Once the kernel confirms to Datacomp that the entire payload of compressed data has been sent (10K in this example), Datacomp may inform the caller that *chunk_size* bytes were sent (32K in this example). At this point, the caller can choose whether to send the remaining data (224K in this example) and the cycle will repeat.

Decompression is the analogous process in reverse. Instead of writing to the DCFD, the DCFD is read from using `dcread()`. `dcread()` reads the encapsulation format of the data (described in the next section) to determine how the data was compressed. Since the Datacomp sender always sends a complete compressed chunk, the Datacomp receiver knows that it can expect to receive the entire compressed payload. Once that payload arrives, it performs the decompression operation and returns the uncompressed result to the application with the return code indicating the uncompressed size.

The semantics of `read()/recv()` are similar to `write()/send()` in that the user specifies a file descriptor, a buffer to read into, a maximum length to read or write, and some optional flags. When a call is made to `read()`, anywhere from zero *up to* the specified number of bytes is read from the file descriptor and copied into the provided buffer. In the simplest form (which Datacomp emulates), the call to `read()/recv()` blocks while waiting for the kernel to provide the data. If there is no data – for example if the end of the file has been reached or the TCP socket is closed – the kernel will return 0 bytes.

As with `dcwrite()`, `dcread()` faces some interesting challenges due to the legacy API. With blocking I/O, the application must be prepared to wait indefinitely for the data to arrive. This works well for Datacomp on the receiving side, because it cannot reliably provide any data until it receives a complete compressed chunk of data ready for decompression and is able to decompress it. However, if an application requests only one byte, Datacomp may be required to read any number of compressed bytes (up to the maximum chunk size) to provide that single byte.

To continue the previous example, Datacomp on the receiving side would need to read the entire 10K compressed payload. This 10K payload decompresses into 32K of raw data, but the application only requested (and potentially has only allocated memory to store) one byte. To obey the API, Datacomp thus provides the singe byte to the caller, buffering the remaining *(32K – 1)* bytes in a structure attached to the DCFD.[52] Subsequent `dcread()` operations on this DCFD will first be served from the buffered uncompressed data. Once that pool is exhausted, Datacomp will continue to read, decompress, and buffer data from the underlying file descriptor.

The approach just described works perfectly in Datacomp's simplified environment and test applications. However, the current prototype would run into problems if it were "dropped into" arbitrary real world programs. This is because DCFDs only support the reading and writing using the default "blocking" semantics; other capabilities are left for future work. For instance, seeking (fast forwarding or rewinding to a particular point in the file) and low-level file coherency are two important features that Datacomp currently lacks but which would not be a problem to implement.

Network streams do not support seeking; once the kernel acknowledges receiving TCP data, it will provide that data to the application in order or throw it away when the application closes the socket. However, files on disk can be changed "behind an application's back." Additionally, applications have random access into files – they can reposition the file pointer

[52] At first, it seems that Datacomp could be at risk for an especially well-compressed input to exhaust memory upon decompression, since it is obligated to buffer any unread data it decompresses. However, because data is compressed in quantized chunks *no larger* than some maximum size, Datacomp can be sure that no chunk of data will be larger than this size upon decompression.

within the file at will. Without enhancements, both of these behaviors would lead to inconsistency in any buffered uncompressed data stored in the DCFD.

To continue our ongoing example, if the user performs a `dcread()` from a file and requests only one byte, Datacomp will decompress the next "chunk" of data, return one byte, and buffer the remaining *(32K – 1)* bytes. Datacomp currently assumes that subsequent reads from this DCFD can be served directly from this buffer and will not notice if the file on disk is changed between the time of the one-byte read and any subsequent reads. If the data is changed, Datacomp would erroneously serve the out of date data from the buffer, while the kernel (without Datacomp) would catch this change as its pagecache was updated by the writes. Another way of describing this issue is that Datacomp decompression implements a kind of file cache which is currently unaware of low-level consistency issues.

To overcome this issue, Datacomp would need to check integrity between the buffer and disk (perhaps using checksums to indicate changes in compressed chunks), or by simply not buffering disk data following "short" reads. Instead of buffering, Datacomp would discard the unread portion of the chunk, rereading, decompressing, and copying the requested subset as necessary. This could work well for file data, since recent file changes would likely be in the kernel pagecache and so could be reread quickly (and this consistency issue is not a problem for network streams). Similarly, while DCFDs do not currently support seeking, if the application read one byte and then performed a `dcseek()` 1K into the file, Datacomp would simply need to be able to update the pointer into the buffer (if applicable) or the underlying file descriptor. Since both of these obstacles are surmountable – and do not add to the research value significantly – they are left for future work.

6.3.2 Protocol Format

Like Comptool, and as alluded to in the previous section, Datacomp cuts the input into sections and compresses each section individually. We refer to these sections informally as "chunks" or more formally as Datacomp Frames (DCFs). Since each DCF is compressed using one of a variety of algorithms and may have a wide range of uncompressed and compressed sizes, Datacomp requires a mechanism to annotate the output so that the receiver can decompress the data. We call this protocol the Datacomp Stream Format (DCSF).

Figure 11 depicts the DCSF. Each frame is composed of a payload that is the result of compressing the input chunk with some method, and a Datacomp Stream Header (DCSH) indicating the information necessary for decompression. Thus, a DCF contains the following elements:

- A Datacomp Stream Header with the following elements:
    - A magic number (0xdatac0bb) indicating the start of a DCF (4 bytes)
    - The length of the uncompressed payload as an unsigned integer (4 bytes)
    - The algorithm and strength used to compress the input (4 bytes)
    - The length of the compressed payload as an unsigned integer (4 bytes)
- A payload compressed with some method (which may include null compression)

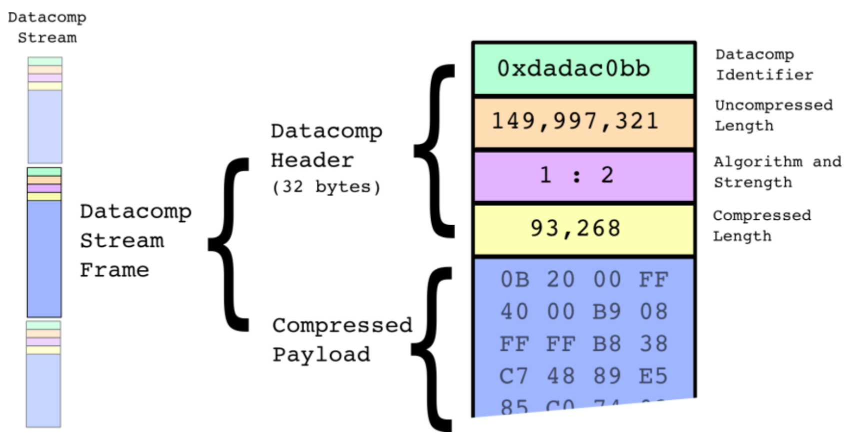


Figure 11. The Datacomp Stream Format.

These pieces of information are used in the following ways. The magic number is meant to indicate the start of a DCF. If Datacomp does not find the correct magic number at the start of a DCF, it will quit with an error. The length of the uncompressed payload is necessary so that Datacomp knows what the size of the payload *should be*, following decompression. This value is also the amount that is returned to the caller of `dcwrite()` as the number of bytes dispatched by the call. While four bytes (an unsigned 32-bit integer) may seem like an arbitrary size restriction, this is the maximum size of an uncompressed DCF – not a limit imposed on a Datacomp file or network stream. Thus, while the largest individual chunk size is currently 512K, the *chunk size* could be as large as 4.3 gigabytes with this header format.

The next value in the header specifies the algorithm and "strength" used to compress the input. This is used to select the appropriate decompression method. For most compressors, the same decompression algorithm is used regardless of the compression options (such as gzip strength levels). However for some compressor families (such as LZO), different "compression levels" can require distinct decompressors. Thus, including the strength in the header is necessary to support decompression in general. The final two elements of a DCF are the length of the compressed payload, followed by the payload itself. Including the size of the payload means that Datacomp does not need to scan the input for a marker indicating the end of the frame. Rather, it can simply consume *payload_size* bytes immediately following the last entry in the header and pass the entire frame to the decompression mechanism.

Again, a fixed-size four byte field may seem like an outdated and arbitrary limit in an era of 64-bit computers. However, the DCSHs are one of the readily apparent sources of overhead for Datacomp, so minimizing their size is more important than supporting unlimited length payloads. Also, the payload size field describes the *compressed size* of the DCF only (not the total input). If any limitation of the header fields is problematic, it would be the uncompressed size, not the compressed size. Because compressing 4.3 gigabytes still takes quite a long time, I believe this is a reasonable compromise.

The elements in the DCSH would be useful for extending Datacomp with random access capabilities for decompression (i.e., `dcseek()`). Since the header includes both the compressed size and the uncompressed size, Datacomp can easily identify which of a sequence of frames contains a specific set of bytes requested by the user. To find and decompress the data in a stream of Datacomp-compressed data, Datacomp would not need to decompress all the data or scan it byte-by-byte. It could instead "stride" through the stream frame by frame until it reached the frame or frames corresponding to the requested uncompressed offsets. Then it would decompress the data and return the selected bytes.

As a final thought, this protocol could benefit from the straightforward addition of a checksum computed over the frame. The result of decompressing corrupted compressed data is undefined at best, and a checksum would facilitate the improvement of POSIX compatibility by efficiently identifying changes between a cached frame and the frame on disk. Compression tools typically use one or more checksums (in addition to the other critical header data specified here) to recognize corruption of file data during transmission on storage, but libraries do not always include them by default in their compressed output. Because TCP already provides data integrity and I did not want to add extra overhead to the protocol for this research, the DCSF prototype does not include a checksum.

6.3.3 Threading and Parallelism

Most previous AC systems emphasize the importance of having separate threads for compression and sending. Additionally, one of the goals of Datacomp was to investigate the underexplored use of parallelism for AC. In this section, I describe how the mechanism of Datacomp uses multiple threads to improve efficiency.

Previous works have stressed the use of separate threads for compression and sending. This architecture is important because it allows the operations to be pipelined – data is sent while the next input is simultaneously being compressed. This helps mitigate variable performance in either channel – compression variability due to data and environment, and network throughput due to contention and other factors. In contrast, if compression and I/O are serialized, then one of the two operations is always waiting on the other. Datacomp uses separate threads both on the sending and receiving sides. When sending, the compressor modules queue data to be sent using a separate sender thread. For receiving, a reader thread fills a buffer with compressed data that is decompressed by separate threads upon request. In both cases, care must be taken to ensure that data is processed in the correct order.

Going beyond the simple threaded architecture described here, compression as an operation is generally ripe for the "divide and conquer" method of parallelization, since a given input can be cut into roughly equally sized pieces and compressed in parallel for a significant speed boost. Interestingly, most previous AC systems do not explicitly perform parallelism, although it is sometimes handled by grid middleware. One reason for this may be that the explosion of commodity multi-core computing is a relatively new development even by AC standards. Another reason may be that, even though compression is amenable to parallelism, most compressors do not support parallelism "out of the box".

Rather than individually modifying the compression methods used for Datacomp, I instead implemented parallel compression and decompression within Datacomp's architecture. In Datacomp, parallel compression works as follows: a buffer of uncompressed data is provided to `thread_launcher()`, the high-level compression "manager" mechanism, along with a chunk size and number of threads. Based on the specified number of threads, `thread_launcher()` then launches one compression Method per thread using the Datacomp wrappers, each of which consume and compress *chunk_size* bytes. Each thread ultimately writes their portion of compressed output into a single buffer, with a locking mechanism ensuring that the chunks are written out in the proper order. At the completion of the parallel compression, the outgoing buffer contains *num_threads* encapsulated Datacomp Frames, ready to be sent. `thread_launcher()` then spawns a sending thread[53] which takes the prepared output and actually makes the low level `send()` or `write()` calls.

The decompression process is slightly different. Nothing about the protocol format specifies if the data was compressed in parallel. This is because the result of an *n-way* parallel compression is indistinguishable (in the DCSF) from *n* serial compression operations using the same method. Thus, while the DCSF tells Datacomp which decompressor to use for each frame, it does not specify how many threads should be used. In the current version of Datacomp, this

[53] Datacomp will only use one sending thread at a time; if thread_launcher() is ready to send data, but there is already data in the process of being sent, thread_launcher() will block and wait. This ensures the integrity of the data being sent and also limits the amount of resources being used, since there will never be more than one (possibly parallel) compression task and one sending task executing at the same time.

choice can be freely made by the decompression side. Typically, the receiver would use as many threads as would improve performance.

For receipt, each thread is responsible for reading one DCF from the actual file descriptor and decompressing it into the buffer of waiting data, called "surplus." Locking ensures that the threads read from the descriptor and write the data into "surplus" in order, while decompression occurs in parallel. Since the DCSF includes the size of the uncompressed data, once a thread has read the header for its frame, it can determine whether there is enough room in surplus for another frame. If so, the thread will start an additional thread (up to some configured limit) once it has obtained the payload for its current frame. Once "surplus" is full, no new reader/decompressor threads are started. Another round of receipt and decompression is started in order to "top up" the "surplus" buffer the next time data is read via `dcread()`.

Another reason for the difference between the sending thread architecture (parallel compress, single send) and the reading thread architecture (parallel read-and-decompress) is that I wanted to be able to measure, discretely, the difference between compress-and-send with one thread and the same operation with two or more threads. This is easier if there is a definite boundary around a single "*n-thread* compression operation." Datacomp starts timing when the first compression thread starts and stops timing when the last byte of that operation is sent. If each thread were pipelined as with decompression, it would be more difficult to calculate the performance of a specific strategy.

On the other hand, with decompression, the choices are much more limited, so I felt that maximizing performance was more important than being able to measure the performance of specific decompression strategies. Also, the lack of constraints means that parallel decompressions can consist of heterogeneous mixes of algorithms – two "nulls," one "lzo" and a "zlib", for example – in other words, there are many more potential multithreaded decompression Methods than compression Methods. This further complicates trying to instrument or store this information. As a result of these factors, each parallel decompression operation is performed and measured individually.

This discussion of timing and measurement leads naturally to the next section, which discusses the "monitors" used by Datacomp in the service of making decisions.

6.4 Monitors

Monitors[54] are the components of an AC system responsible for obtaining and making available information about the environment and data for use in AC models. Datacomp uses monitors to read and present data about Event Results, the CPU load and frequency, available bandwidth (ABW) and to predict the general compressibility of input. Datacomp also provides facilities for timing various events. The information from all these sub-systems is used both directly and indirectly in the process of making compression decisions.

6.4.1 Timing, Buffer Sizes and Output Rates

Two of the most important resources that Datacomp tracks are the time required to perform various operations and the size of compressed and uncompressed data used in those operations. However, unlike the other explicit and relatively self-contained monitors described later in this section, this instrumentation is placed at judicious intervals throughout Datacomp's code. While many discrete operations (such as the time to open and close various resources) are instrumented, the most important measures are the payload sizes and time costs of compression and transmission.

[54] Described generally in Section 3.3.3.2 and discussed in the literature in Section 4.2.3.

While the Opportunities (the combination of Environment and Data) are not defined by timing data and buffer sizes, the Results of specific compression Events are. Datacomp uses observed timing data and payload sizes to compute the output rates of various operations. Then, when optimizing for time or energy, Datacomp makes choices based on environmental monitors with the goal of maximizing the overall output rates as measured by the timing infrastructure.[55]

Compression output sizes are usually stored in a variable by the underlying compression library (e.g., zlib). Other internal Datacomp buffers have pointers that enable quick size readings. Virtually all timing operations in Datacomp are performed using the same instrumentation mechanism (a generic timing tool I built into Datacomp), with timing points wrapping the generic operation being timed whenever possible. In this way, every event is timed in a uniform fashion at the same points in the code, even if the code performing the event is different (e.g., if different compressors are being used).

6.4.2 CPU Load and Frequency

Datacomp monitors CPU load and frequency using a utility called `datacomp_cpumon`. This CPU monitor is implemented as a separate application that can run for long periods of time in order to avoid adding additional start/stop costs to Datacomp, and because the information is global and not related to any specific Datacomp process. Another way of looking at it is that, rather than being tightly coupled to a single Datacomp instance, the data provided by `datacomp_cpumon` is useful for multiple instances of Datacomp. The capabilities of `datacomp_cpumon` could easily be provided by the OS and would ideally be integrated into an AC-using operating system. Implementing `datacomp_cpumon` as a standalone program is a compromise towards that ideal.

[55] When optimizing for space, Datacomp ignores the time cost.

`datacomp_cpumon` is started from the command line and takes only one parameter – the sample rate at which the tool publishes the CPU statistics. `datacomp_cpumon` uses information made available in the `/proc` and `/sys` file systems. These are special files containing plain text system information exported by the kernel, and are world-readable. Specifically, `datacomp_cpumon` uses per-core timing information made available in `/proc/stat` that can be used to compute the average load for that core over the timing interval. This is a more accurate and useful value than the standard UNIX load average. `datacomp_cpumon` uses CPU frequency information available in `/sys/devices/system/cpu` to determine the maximum frequency and current per-core frequency. In addition to reducing start/stop overhead, a persistent `datacomp_cpumon` is able to sum and average the CPU load and frequency over a short period of time which smooths the output.

For example, executing "`datacomp_cpumon 10000`" will start the monitor running in its own process, sampling the requisite system facilities and computing the summary statistics up to 10,000 times per second. The resulting statistics are published in the text file `/datacomp/dc.cpustats`, which includes two numbers: the mean CPU load and the average frequency across all the cores. Both numbers are stated as an integral percentage; for example, the data "`40 33`" indicates that the mean CPU load is 40%, and the mean CPU frequency is 33%.

The file `/datacomp/dc.cpustats` is world readable. Thus, any software on the system that wishes to use these figures can read them directly from the file at whatever frequency is desired. Again, while Datacomp (or any other application) could read the original sources of data and compute the averages themselves, putting this information in one location makes it available to other applications or other Datacomp instances. Since this information is independent of any particular application, having multiple programs computing these values independently would be inefficient.

Since `datacomp_cpumon` uses a file to communicate its findings, it is important that Datacomp minimize the cost of file access when it needs to determine the current CPU load or frequency. When a DCFD is created using `make_dcfd()`, a thread is started whose sole purpose is to read the file `/datacomp/dc.cpustats` on a periodic basis by seeking to the beginning, reading the values, and sleeping before repeating the process again. Every time the values are acquired, a mutex is taken and the statistics are written into the environmental structure portion of the DCFD. When Datacomp proper needs to access this information (e.g., to make a decision), it acquires the mutex, reads the statistics, and releases the mutex. In this way, the cost of reading the CPU properties is the price of a mutex, rather than a system call and the cost to parse the underlying file. Since only two threads use the mutex and the operations they perform while holding the lock are small, lock contention overhead should not be a significant issue. Also, since this is the only lock acquired in this portion of the code, deadlock over the resource is not possible.

6.4.3 Available Bandwidth

The success of Adaptive Compression hinges on being able to perform compression quickly and effectively enough to improve the throughput of the I/O channel in question. If an adaptive system is to be able to make profitable choices, it must have a reasonably accurate estimate of "how fast" the I/O channel will be during the short period of time when the choice will be in effect. This is complicated by the fact that network throughput can fluctuate significantly over long or short time periods due to many factors, such as network utilization, contention and interference.

As a result of the fluctuating nature of real networks and the need for the throughput statistic to be useful in the short term, more easily obtainable characteristics of the network are not suitable for making compression decisions. Instead, the critical value necessary for making profitable choices is the *current available bandwidth* (ABW) – the effective data rate of the network at the time of use.

At first, acquiring this value seems straightforward. It might seem that a good first approximation is the fixed bandwidth of the physical layer. However, not only might this value not be readily available for some types of I/O channels (e.g., the sustained throughput of hard disks), but it fails to describe changing conditions, which are critical aspects of AC systems. Suppose a system is attached to a 1gbit/s LAN; emitting compressed throughput at this rate currently requires special algorithms or incredibly fast hardware. A system estimating network throughput purely on the physical layer might never use compression, even if contention causes the *available bandwidth* to decreases to levels where compression can be often be useful.

Thinking about the problem more deeply, we recognize that there are at least four significant problems facing a user-space application needing to know the current ABW of a channel without access to any third-party monitors, like NWS [62]. The first and most obvious problem is that the available bandwidth is dependent on factors that the independent system cannot know or control, particularly if the network is involved. As a result, the best we can hope for in practice is to estimate the available bandwidth.

The other three problems are less obvious. One is that user-space applications are not in control of low-level I/O. Instead, users perform system calls, submitting buffers and lengths to the kernel, and the kernel performs the tasks on behalf of the user, returning values indicating the results. In the process, the kernel obscures many of the details necessary for calculating an accurate ABW estimate. This abstraction boundary poses an obstacle to transparency. As a result, developers can try to be cleverer in their approach to estimating ABW, but in the process of doing so, they run into the two remaining problems: buffering and scheduling.

Buffering is a problem for calculating available bandwidth for many of the same reasons it is difficult to accurately simulate I/O channel throttling for Comptool (as described in Section 5.3.3). An obvious approach for estimating ABW is to measure the time spent in `send()` or `write()` calls and combine this with the amount of data involved. However, the kernel buffers these events. As a result, write events that do not fill the kernel buffer (and reads in general) can be completed – from the user's perspective – in the tiny amount of time it takes to serve a system call. For example, a `write()` of one megabyte on a 100Mbit/s channel that takes 0.002s to complete (because the entire 1M fits into the buffer) translates to an incredible throughput of 500 *megabytes* per second – when it should not be any faster than 12.5M/s at the most.

While throughputs calculated from individual system calls can settle on the true available bandwidth, this only occurs over a long period of time or if the buffer is consistently full (resulting in making the "sustained throughput" visible). If the buffer is large, the size of the average payload being written is small, there are delays between write events or the available bandwidth of the channel is great, it can be difficult to keep the buffer full. As a result, average throughputs of individual I/O events can often be unrealistically high. This can pull up the average enough to meaningfully overestimate the available bandwidth. For AC, this leads to a reduction of compression, because there are fewer methods capable of "keeping up" with the network as throughput increases.

Instead of measuring the data rate at the system call level, a different approach is to simply sum the number of bytes of data sent and divide it by the overall run time. This effectively smooths out the fluctuations due to buffering and provides an accurate estimate of the output rate of the application. Unfortunately, the fact that this approach relies on an average over a long period of time means that the statistic can be unreliable early on and is not very responsive to changing network or system conditions later – two properties that are necessary so that AC can be as responsive as possible – capitalizing on short-lived opportunities and avoiding any no-win situations.

A compromise approach is to use a “window” – a number of *n* events which are saved and averaged together. As new events occur, old events disappear from the window. This is an improvement in the sense that the statistic will more closely reflect the current state of the channel. Unfortunately, shorter windows, which are more responsive to network changes, are more likely to result in inflated estimates due to the buffering behavior of the kernel. Larger windows are, again, less responsive.

Most unfortunately though, the output rate of the *application* – the rate at which the application is able to produce and send data through the kernel and perhaps the most obvious source of data – is not the statistic we need. The application’s “output rate” conflates the rate at which the program itself generates data – which with AC is *compressed* – and the rate at which the network can consume that data. This can lead to the program mistaking its own variable performance (and the effect of AC) as reflecting the state of the network. For example, compressing data with `xz` will almost certainly reduce the average output rate of the program. But it could increase the average throughput of the overall communication.

Probes or a side channel could be used to communicate timing information between two parties. In this way, the parties could determine how much time elapses between when some message is sent and when the receiver obtains the data. However, this approach doubtless has its own challenges and pitfalls. In this work, I did not pursue this approach primarily because the use of side channels was somewhat outside the Datacomp design philosophy.
If the fundamental obstacle to obtaining an accurate estimate of the ABW is that the kernel obscures critical pieces of information, then access to kernel information seems like a likely solution. While a kernel-based AC system would have easy access to this data, the standard POSIX API for networks does not provide this level of insight into the kernel.

Certain operating systems provide alternative means to acquire some of this information using the special system call `ioctl()` which stands for Input-Output (IO) Control. A call to `ioctl()` is parameterized with the file descriptor of the I/O channel in question and the integer value corresponding to the request being made, and returns the value corresponding to the request. Both BSD Unix and Linux provide `ioctl()` requests that query the kernel to find out how many bytes of data are unacknowledged in the send queue for a particular socket. For BSD, this `ioctl()` request is FIONWRITE, while for Linux the request is SIOCOUTQ (these "human readable" names are macros stand in for the correct request number).

Armed with this `ioctl()` request, Datacomp can track the number of bytes it writes to the underlying socket, measure the elapsed time, and then query the send queue to discover how much of that data has been sent in the intervening period. With care, this can be used to acquire a reasonably accurate estimate of the recent network throughput. I say "with care" because it is not simply a matter of performing a write, waiting *n* seconds, and determining how much data has drained. Waiting too long will result in not seeing the changes in the queue length, because data might be fully drained between checks. On the other hand, checking too often will result in many tests where the queue length does not change, making the ABW appear to be zero, because the kernel services the sockets one at a time.

At a high level, Datacomp's ABW estimation mechanism works by tracking the time and amount of data written to the socket at each low-level `send()` event while in parallel regularly (e.g., 100 times a second) querying the system time and send queue length using the `ioctl()` call. When the queue length decreases significantly (a configurable amount), Datacomp recognizes that the kernel has served the socket. It then determines how much data had been written to the queue since the last time the socket was drained, how much data was just drained, and how much time has elapsed. These figures are then used to calculate the bandwidth that would have allowed that transfer to complete. The resulting ABW estimate (called an Estimated ABW or EABW) is written into the DCFD structure where it is available to Datacomp modules that need this estimate (protected with a lock similar to the CPU load and frequency mechanism described in Section 6.4.2).

In this way, the EABW reflects the most recent period of time for which a meaningful bandwidth can be computed. If no data is being sent, the EABW figure stays at its most recent value. In order to bootstrap the EABW mechanism, Datacomp saves the last EABW value to a file when Datacomp closes. When a new DCFD is instantiated, Datacomp loads this value as the starting EABW.

6.4.4 Compressibility

As discussed in Sections 3.3.3.2, 4.1.2 and 4.2.3.3, most AC systems either explicitly measure or estimate compressibility using some technique, or at the very least decisions are ultimately based on compressibility in some way, due to the importance of compression ratio in determining the value of an AC system.

As discussed in Section 4.1.2, calculating the compressibility of input data is a difficult proposition for AC systems, because the only perfectly accurate method to predict the compressibility of an input with a method is to actually perform the compression with the Method in question. This is almost always too costly: data is not always compressible, choices are often made between *multiple* compression Methods and time is usually of the essence. As a result, if explicit predictions are made, they are typically based on heuristics. For example, ACE [62] uses the "last block assumption," which assumes that the next quanta of input data will be as compressible as the previous chunk. This approach will result in errors when compressibility changes quickly.

Other systems compress a small *sample* of the input [64] [77], using the resulting compression ratio as an estimate for the total. However, because the accuracy of this approach depends primarily on how much data is sampled,[56] greater accuracy requires greater investment of time. Additionally, since compression time depends on the characteristics of the data, the overhead of sample compressions could be variable, which complicates the cost-benefit analysis of choosing when and how to sample. On the other hand, sampling could be useful in the sense that it can provide both compressibility and time estimates.

In an effort to respond more quickly to potentially disparate types of data and to avoid using actual compressors, Datacomp uses an explicit prediction technique that is similar to but was conceived separately from work described in the undergraduate thesis of William Culhane [48]. I refer to these kinds of heuristics as "bytecounting" approaches. In bytecounting, some

[56] And, if appropriate, the accuracy of your model relating the achieved CR of one method in a given opportunity with that of another.

statistical summary of the input is taken over the data (e.g., at the byte level) and is used in some way as an estimate of the compressibility of the data. This process is typically inexpensive and the cost should not vary by data type as significantly as actual compression operations would. Furthermore, like test compressions, bytecounting can be performed on a sample of the data. This can be valuable if the bytecounting approach is too costly for the full input.

Bytecounting approaches estimate the “evenness” of the input data [48]. Consider a naïve approach called Checkbox. Checkbox simply keeps track of how many distinct 8-bit bytes appear in an input, “checking them off a list” if they appear at least once. If a one-megabyte input only contains one unique byte, or even two, it is bound to be highly compressible because the number of unique sequences of bytes of some fixed length would be extremely limited – and would provide an excellent opportunity for compression. However, if Checkbox discovered that in that one megabyte of data, all 256 distinct 8-bit bytes appear, there is no way to know whether the data is completely random, 256 bytes followed by zeros, repetitions of a single 256-byte pattern, or anything else. Thus, useful bytecounting approaches generally include some mechanism to help tease more information out of the data. For example, two of Culhane’s heuristics include some information about byte sequences by incorporating statistics computed across pairs of sequential bytes.

Another important property of a bytecounting heuristic for the purpose of AC is how it handles false positives and false negatives. Due to the high cost of compressing uncompressible data (especially at LAN speeds), it is more important to not compress uncompressible data than it is to always compress every piece of compressible data. In other words, we would prefer a mechanism that has fewer false positives than false negatives.

The approach I used for Datacomp is simple. In short, it attempts to put a number on the evenness of the data by counting the number of bytes that appear as frequently or more than they would if the bytes were distributed evenly. If any byte appears at least *threshold* times in the input, it is counted as being "overrepresented." The threshold is computed by dividing the input length by the number of possible unique bytes (i.e., 256). The total count of these "overrepresented" bytes is the output of the mechanism; higher values show that a greater number of bytes are closer to being evenly distributed, which would be true if the data contained a significant amount of non-ASCII characters, was already compressed, encrypted, or otherwise close to being randomly distributed. Conversely, compressible data (like natural language) must have some kind of repeating structure, which tends to result in certain bytes appearing much more often than others.

```
appearances[256] = 0  # an array holding the appearance counts of each byte
bytecount = 0  # the total number of "overrepresented" bytes in the input
threshold = length(input) / 256  # the threshold value for this run

for byte in input:

      slotnum = int(byte)

      if (appearances[slotnum] < threshold): # if byte not overrepresented

            appearances[slotnum]++  # add one to the count for this byte

            if (appearances[slotnum] == threshold): # if threshold reached

                  bytecount++  # increase total bytecount

return bytecount
```

Figure 12. Pseudocode for Datacomp's bytecounting heuristic.

Figure 12 is a pseudocode representation of Datacomp's bytecounting algorithm. First, an array of 256 values is created and initialized to 0, the total `bytecount` (the return value) is initialized to zero, and the threshold is computed by dividing the length of the input by the number of possible bytes.[57] Next, each byte in the input is examined. The byte is converted into its 8-bit unsigned integer value and assigned to `slotnum`, which is used to index into the array `appearances[256]`. If the count for `slotnum` is still below or equal to the threshold, it is increased by one and compared to the threshold. If it is now greater than the threshold, the total `bytecount` is increased by one, and future occurrences of this byte will not be counted.[58] Finally, once all the input has been scanned, the total number of "overrepresented" bytes (i.e., `bytecount`) is returned.

It is important to emphasize that this is simply a heuristic for identifying data with different compressibility properties and does not actually measure compressibility. Again, a repeating sequence of all 256 distinct bytes would have a high "bytecount" using my method even though it would be eminently compressible. However, not only does this kind of data appear to be rare, but I consider this a "feature," not a "bug," since it shows that the mechanism "errs on the side of caution" – identifying compressible input as uncompressible, rather than the other way around. Discussion of the bytecounter in practice is in Section 10.5. While the bytecounter may sometimes mistakenly estimate that uncompressible data is compressible, it is accurate enough to be generally effective for my purposes.

Another issue to keep in mind with bytecounting is that it is vulnerable to the same issues as sample compression; if an unrepresentative sample is taken, an unrepresentative result will be returned. For example, if a JPEG image is sampled, but the sample data happens to be drawn from the EXIF metadata (which is plain text), the bytecount of the sample will probably suggest that the sample is compressible. However, one of the key values of bytecounting is that it is

[57] This can be generalized to other "byte" lengths, but Datacomp uses 256 slots which is also easy to read for those of us raised on 8-bit bytes.

[58] These extra tests (only incrementing if the count is still below the threshold, etc.) are time optimizations meant to skip the work of tracking bytes which have already crossed the threshold.

inexpensive in comparison to compression. As a result, it can be affordable to use bytecounting on more data than would be feasible for sample compression.

The base bytecounting code requires approximately 0.002s per 512K on my primary testing system. This is about half the time required to perform LZO compression on the same data. Since LZO compression is actually useful (if you could send the compressed output), it is interesting to consider the possibility of using LZO as an efficient estimator itself. However, there are some drawbacks to this idea, which I discuss in the Datacomp evaluation. Additionally, while this overhead is insignificant at low bandwidths, it can become significant at higher bandwidths, particularly since it is paid for all data regardless of what compression method is chosen.

In order to reduce the cost of bytecounting, the current system samples 1/10th of the available input, up to the maximum chunk size. It performs sampling by "striding" through the input, sampling 1K and skipping a portion of data based on the input length before sampling again until the entire input has been scanned. In this way the total number of bytes "counted" totals 1/10th of the input. These individual counts are averaged (rather than totaled), which retains some of the local information gathered in each sample. With this scheme, each bytecount operation costs approximately *0.0004s* per 512K.

Ultimately, the goal of Datacomp is to show that locally independent AC is practical and can be effective on real world workloads. One way that I proposed to do this was through the use of "efficient estimators" for compression. Bytecounting is one of these efficient estimation techniques. At the same time, I do not believe that my approach is the most accurate or efficient bytecounting mechanism or compressibility estimator. However, it is simple, inexpensive and it serves its purpose well. Analysis of the accuracy of my bytecounting approach can be found in the results sections.

6.5 Models

In Datacomp, "models" are the components responsible for making compression decisions. Each DCFD has a function pointer – `decide()` – that points to the Model code responsible for optimizing the resource that was specified when the DCFD was instantiated. When an application calls `dcwrite()`, the relevant monitors are read, and whatever Model the `decide()` function pointer references is executed. This Model makes a compression *choice* based on the available information and launches threads responsible for performing any further compression (if necessary) and transmitting the data. The Model returns the number of bytes consumed (submitted for compression and transmission) to `dcwrite()`, which returns to the user.

Depending on the design and purpose of the model, it may test the compressibility of the input, perform test compressions (when optimizing for space), obtain data relating to the opportunity that was not obtained by `dcwrite()` (e.g., by "bytecounting" the input), consult actual predictive models, and more. Since most of the AC system's state is available in the DCFD structure, the Model code is quite free to make decisions in any number of ways. In order to focus the responsibilities of the Model on making decisions, Datacomp's Models are not responsible for performing the compression operations.[59] Rather, they inform the Mechanism of the Method choice, and the Mechanism carries out the choice of the Model. This section will discuss the design and behavior of the Models constructed for this first prototype of Datacomp.

[59] The "space" decider is something of an exception to this rule, but the exception serves to illustrate the flexibility of Datacomp's internals.

6.5.1 Manual

If the manual Model is specified when the DCFD is created in the initial call to `make_dcfd()`, the DCFD function pointer `decide()` will be set to the function `manual_decider()` (which I will refer to as MD), defined in `manual.c`. MD is hardly a "model" at all. Instead, MD simply performs some universal housekeeping tasks and dispatches whatever compression method is specified in the DCFD's "outbound method" structure. In the test applications `dzip` and `drcp`, these variables are set by command line options in order to specify a specific compression method. In other words, MD is a base case for testing and comparing Datacomp's raw performance when using a static compression strategy. However, MD is useful to discuss as a baseline for the more complicated deciders because it includes many of the basic features common to all Datacomp models.

When MD first starts, it reads the values of the variables specifying the manual compression method. Next, MD determines whether there is enough input to compress with the specified mechanism. Datacomp always automatically uses null compression (regardless of the manual mode) if the length of the input is less than *DC_COMPRESS_MIN * numthreads* bytes, where `DC_COMPRESS_MIN` is set to 512 bytes in the current system. This is to help Datacomp avoid wasting resources, since extremely small inputs are unlikely to benefit from compression, especially from a throughput perspective. The value *numthreads* is included so that the model can verify that there is enough data for each thread to compress. If there is enough data to compress, MD will consume *(chunksize * numthreads)* bytes, or all of the available input, whichever is less.

While `DC_COMPRESS_MIN` is a "hard coded value," which I have attempted to avoid in the design of DC, it can be changed in the source code. Furthermore, it is intentionally set extremely low (especially compared to other AC systems). This value is not meant to ensure that data will be compressible, but rather to avoid situations where compression is almost certainly a waste.[60] In this way, Datacomp tries to avoid compressing in hopeless situations without being overly conservative.

Once the amount of data to be compressed is determined, MD bytecounts up to the amount of data specified by the manual mode (unless statistics collection is disabled). The bytecount and other statistics are optionally collected in many components of Datacomp for the purposes of testing and design and can include such elements as compression method, timing information, CR, bytecount, bytecount time, and so on. Statistics, if collected, are typically written out to a log file for offline analysis (unless this is disabled for testing). Finally, MD calls the DC function `thread_launcher()`, which executes the compression method and sends the results, as described in Section 6.3.3. Once `thread_launcher()` returns from starting the threads, MD returns to `dcwrite()` the number of uncompressed bytes from the initial message that were compressed and sent.

6.5.2 Space

The basic outline of `manual_decider()` holds for `space_decider()` (SD), with some specific changes due to the nature of the brute force compression search mechanism. Like MD, SD is called from `dcwrite()` if "space" was specified as the priority when the DCFD was created. While MD simply reiterates a user-specified method, SD makes a dynamic decision based on the characteristics of the input data and the configured compression methods.

There is no obvious method for determining the most effective compression choices ahead of time. One approach could be for Datacomp to consume all the input and exhaustively

[60] This is not to say that no compressor can compress 512 bytes of data, but rather that a message of this size is probably not worth compressing.

try every possible combination of choices across the *entire input*. While this approach could be practical for small inputs, it would generally be impractical since it could require many passes over an arbitrarily large amount of input.

This kind of approach is also somewhat against the design philosophy of the POSIX interface that Datacomp emulates. While the optimal solution could be determined for the payload of a single call to `dcwrite()`, to truly "brute force" the ideal strategy for a file or long transmission (which may require many calls to `dcwrite()`), Datacomp would need to consume all the input the application intended to send or write, without actually sending or writing anything. If an application cannot or will not write all the data for a given task at once, Datacomp would have to "lie" to the application – falsely claiming that some data had been written in order to get the next portion of input. This would break the API and could cause major errors.

Because of these constraints, Datacomp's Space Decider uses a greedy heuristic (virtually identical to Comptool's) to determine the "best" compression strategy. Specifically, SD performs all combinations of compression methods starting at offset 0 in the `dcwrite()` payload (this includes varying the chunk sizes). Unlike MD (and the Time Decider described later), SD actually calls the compression operations *itself* (using the API wrappers) in order to perform the greedy heuristic. While Datacomp does not include any models that perform "sample compressions" on subsets of the data, this ability demonstrates that models do have access to the compressor functions that would be necessary for a sampling model.

After SD performs each method, it calculates the compression ratio achieved and other performance statistics, such as the compression time. All the results from the event are saved in a structure in SD's scratch space, including the compressed output. If this is the first event, this structure is automatically copied into a structure holding the "best" method and results. If this is not the first event, but has more desirable properties than the currently saved event, SD swaps the current event state with the saved state. It then continues through the rest of the methods.

Once all the methods have been performed, the result stored in the "best state" structure is submitted to `thread_launcher()` for *immediate* sending (skipping the compression phase). In this way, SD saves `thread_launcher()` the cost of repeating the best compression operation. While time and energy are not particularly important to SD, this ability to submit compressed data from the model exists so that potential sample-compressing models (intended for speed) can reduce redundant compression operations.

Compression ratio is not the only result that SD considers when making decisions. If two or more combinations have the same CR, the tie is broken first by keeping the combination with the largest input size (if applicable) and second by keeping the combination with the greatest output rate (input size / compression time). In this way, the Space Decider (like Comptool) automatically considers run time as a secondary priority.

This means that if a 32K chunk of input at some offset compresses more successfully (has a lower CR with some method) than a 512K chunk at the same offset, the compressed 32K chunk will be used and the pointer into the write buffer will be advanced by 32K (since only 32K of input was ultimately consumed, compressed and written). `dcwrite()` will return to the caller, indicating that 32K of data was written. If the caller has more data to write, it will (as per the POSIX API) will continue to call `dcwrite()` until it has finished. The end result of this approach is that at each opportunity, Datacomp will have chosen the most effective compression method for the given input (up to the largest chunk size currently being used).

6.5.3 Time / Energy

The function `time_decider()` (TD), is the model responsible for optimizing throughput to minimize total transmission time and energy. TD is by far the most complex model used in Datacomp. Its decisions depend on the input of every monitor described in Section 6.4 in conjunction with a locally constructed, persistent and continuously updated history model.

In a nutshell, TD quantizes the outputs of Environment Monitors, creating a manageable set of unique Environments. It does the same with the output of the bytecounter, which results in a reduced number of unique Data types. By combining the set of quantized Environments and Data types, we can identify a manageable set of unique Opportunities. To model these Opportunities, TD tracks the time performance Results for every Event (every use of a particular Method in a specific Opportunity). This allows TD to pick the historically best Method for every Opportunity it encounters. Finally, TD updates the history database with the resulting outcomes of its choices, allowing the database to "learn" over time.

I make no claims that this is an optimal decision model. However, it is a functional model that substantially meets the design goals of Datacomp: it is locally-independent, relies on very few hard coded assumptions or limiting mechanisms and it is fast and accurate enough to be effective in a broad range of circumstances. It is also not inevitably subject to the "death spiral" seen in other AC systems (described in Section 4.2.4.3), because it does not assume any hierarchy among its supported Methods and is able to differentiate between different types of Data and Environments (i.e., Opportunities). This means that it will learn that some Methods are losing propositions for certain Opportunities, even if they are profitable for ostensibly similar situations.

Even though TD is effective, it is not perfect. Two empirical points of criticism include the choice of quantization values (which may not be optimal) and the potential error inherent in using bytecounting to divide data into separate types. Other more pragmatic concerns include the length of time necessary to train the database, its size, and so on. Ultimately, as with the trade-offs used in many AC systems, optimal accuracy, efficiency, and simplicity are in some sense competing goals. The remainder of this section will discuss the design and details of the Datacomp Time Decider and its basic operation.

6.5.3.1 Monitors and Quantization

The manual and space deciders do not base decisions on environmental properties such as CPU load, but instead apply a preselected Method or search for the best compressing Method through trial and error (respectively). In contrast, TD incorporates every monitored source of information described in Section 6.4. The output of every Data and Environment Monitor is quantized into a fixed number of discrete levels, which at a high level drive the selection of compression methods.

As described in the overview, TD relies on quantization (or "bucketizing") of environmental parameters to cope with the practically infinite number of potential Opportunities. This quantization ultimately determines the size and granularity of the history database, because the quantization levels define the dimensionality of the history mechanism. However, the specific quantization methods used and the general design of the history database are not tightly coupled to each other or to Datacomp's mechanism in general. While the database size is defined by the number of quantization levels, this number can be trivially changed in the source code. Additionally, each property can be quantized with a different function, meaning that it is easy to explore the benefits and drawbacks of different quantization schemes. In other words, while quantization as a means for coping with complexity is hard-coded into the design of *this* time decider, the specific mechanisms used to quantize are easily modifiable. And of course, other models are free to make decisions any way they see fit – there is no requirement to rely on quantization.

Nevertheless, the way that a range of a parameter (e.g., CPU load from 1-100%) is quantized represents a kind of hard-coded knowledge. However, it is a practical compromise between the impossibility of treating every Opportunity as unique and the overly simplistic approach of assuming that various factors do not impact the best compression Method (e.g., assuming that data type is irrelevant). The quantization mechanisms, while tuned somewhat empirically, are meant not to direct choices in a particular way based on my expertise, but rather to capture the greatest amount of diversity in the system while still remaining manageable.

In practice, some of these environmental parameters may not be critical. In fact, I believe it is likely that in the same way that more accurate and efficient monitors could improve the performance of Datacomp, different quantization levels and functions could improve the accuracy of the model's decisions. I will discuss some results relating to this in a later section. However, improving these choices or creating dynamic mechanisms to handle this is left for future work. In the remainder of this section, I describe the specific monitors and quantization mechanisms used for the TD.

TD uses the CPU load and frequency as provided by `datacomp_cpumon` (as described in Section 6.4.2). This data is critical because compression requires computation and the number of operations per second that can be spent on compression is dependent in part on both the CPU load and the current operating frequency. `datacomp_cpumon` provides these values as integers between 1-100. There are 10,000 (100^2) possible combinations of CPU load and frequency alone, without considering the other monitored values such as data type.

To reduce this complexity, CPU load is quantized into *N* levels, where the N^{th} level is 100% load and the *(N-1)* levels divide up the remaining range of the load. With four load levels (the current number of levels), the ranges break down as shown in the *CPU Load* column of Table 15. The choice for such large ranges of CPU load with a single value representing 100% load was driven by visible impact of total CPU utilization in conjunction with wanting to map the general effect across the entire load range with as few values as possible.

Level	CPU Load	CPU Freq.	EABW	Bytecount
0	1-33	1-25	Unlimited	>100
1	34-66	26-49	1-768kbit/s	67-99
2	67-99	50-74	768kbit-2mbit/s	34-66
3	100	75-95	2mbit-20mbit/s	1-33
4	NA	95-100	20mbit-200mbit/s	
5	NA	NA	>200mbit/s	

Table 15. Time Decider quantization levels.

CPU frequency values are broken into five levels using a mechanism similar to the quantization used for CPU load and are shown in the *CPU Freq.* column of Table 15. The "maximum level" represents frequency percentages between 95-100%, with the remaining four levels being roughly evenly divided across the range. The intuition between these levels is similar to the choices made for CPU load in conjunction with the fact that the per-core frequency capabilities of modern processors can result in a larger range of frequency "percentages" than older CPUs (which typically only had a few discrete levels). I wanted to be able to capture the effect of these levels in case it was significant.

Estimated Available Bandwidth (EABW) is the final opportunity parameter considered by TD. EABW is provided by the EABW monitor as a rate in bytes per second as described in Section 6.4.3. This value is quantized into six levels, as shown in Table 15. Level 0 is a special case used to calculate the raw output rate of a particular event, resulting when the EABW submitted is 0. The other levels correspond roughly to our test levels of 1gbit/s, 100mbit/s, 6mbit/s, 1mbit/s, and 500kbit/s. These levels were chosen to correspond to common network throughputs for gigabit Ethernet, 100-megabit Ethernet, home cable or DSL connections, fast cellular data and slow cellular data, respectively. However, I manually adjusted these slightly to help provide meaningful differences based on the observed effective rates of popular compressors on various machines.

Specifically, many of the Methods used have maximum performance values that fall between our experimental test levels. For example, on a variety of systems I have used (not just my test system), gzip can easily "keep up" with a 10mbit/s stream of data and can sometimes exceed 100mbit/s, but cannot typically reach 1gbit/s – even with parallelism (with the current implementation, hardware and input). If I divided the EABW levels sharply at 100mbit/s, LZO would dominate the range from 100mbit/s through 1gbit/s, drowning out the fact that gzip *can be* competitive in lower portions of that range. Similarly, heavy-duty algorithms like xz and bzip2 struggle to compete at modern LAN speeds but can perform well on slow networks. By setting the quantization breaks at 200mbit/s and 20mbit/s (for example), I hoped to provide a range of choices that reflected the general performance capabilities of the methods being used. Additionally, the EABW mechanism is conservative and has had a tendency to underestimate the actual available bandwidth. By increasing each threshold a bit over the corresponding test level, I hoped to compensate for this error.

As with CPU load and frequency, these quantization levels reflect a manual choice based on human expertise. However, like them, the choices were made to maximize the power of the Model, rather than to bias the Model towards particular Method choices. Ideally, quantization levels for EABW, CPU load, and frequency would be able to tune themselves to maximize the power of the model. This is left for future work.

Bytecount is used as a proxy for broad compressibility. When new data is submitted to TD, up to the maximum chunk size is bytecounted. With the current bytecounter (described in Section 6.4.4), the range of possible values is from 1 (the entire input is composed of the same byte) up to 129 (the maximum number of bytes are "overrepresented"). This represents yet another combinatorial headache, and so it is quantized into four levels as shown in Table 15. In my testing, a bytecount of 100 or more (level 0) is very unlikely to be compressible. Bytecounts below 100 (levels 1-3) are generally compressible, with lower bytecounts being more compressible. Because wasting time on fruitless compression is a total loss for Datacomp, it is critically important to avoid these situations. Thus, the TD assumes that data with a bytecount greater than 100 is uncompressible and does not look up a method or store the results for that kind of data.

This assumption could be considered hard-coded knowledge in the sense that some data with a bytecount greater than 100 may be compressible. However, the vast majority of such data is not compressible according to my results. Thus, the end result of tracking this data in the database would be an entire section of the database – complete with computational, space, and learning time overhead – dedicated to permanently recommending "null compression." In any case, like the quantization of the other monitored properties, the number of levels, the quantization mechanism, and even the bytecounting monitor itself are all easily changed.

6.5.3.2 Computing Estimated Effective Output Rate

In the same way that measures such as the average throughput (Section 6.4.3) or the run time of compression (Section 5.4.2.4) are not the important measures for AC decisions, calculating the statistic of interest for maximizing time is a bit counter-intuitive. In Section 5.4.2.4, I explained how Comptool calculates the Effective Output Rate (EOR) and chooses the compression events with the highest EORs in order to minimize the overall transmission time. EOR can be calculated as:

$$EOR = uncompressed\_size / operation\_time$$

Abstractly, this equation is still true for Datacomp, where *uncompressed_size* is the size of the input data and *operation_time* includes both the total time spent in compression and transmission. However, unlike Comptool, which performed operations discretely and in serial, Datacomp performs the operations in parallel and writes to real files and network sockets – complete with buffering effects. This means that the same "event" can take very different amounts of time because of system effects. Just as this effect makes it difficult to calculate an ABW estimate, it makes it difficult to calculate accurate mean performance values when looking at individual, short-duration events.

To avoid this issue, we need a different way to estimate the effective output rate (EOR) of a given event. This calculation should avoid relying on the "syscall time" of the internal `send()` or `write()` calls. On the other hand, compression execution time and achieved compression ratio statistics are reliable, but they vary from input to input.

It might be tempting to instead use the output rate of the compressor instead of the EOR as described above. This "algorithm output rate" (AOR) can be calculated as:

$$AOR = input\_size / compression\_time$$

However, this value fails to include the benefit of compression in that it ignores the compression ratio of the method which allows us to use the ABW more efficiently. The effect of using AOR is that, regardless of the bandwidth of the environment, high speed algorithms such as LZO (or even "null compression") will be extremely hard to beat, even though they do often not compress data as well as other methods.

Another tempting approach is the effective uncompressed rate of the compressor. We can calculate this as follows:

*EUR = (output_size / compression_time) * compression_gain*

This statistic measures the rate at which the compressor emits compressed data, multiplied by the compression gain (the inverse of the compression ratio or "expansion factor"). This shows (not including decompression time) the rate of information flow. It allows "slow but strong" algorithms to be compared against "fast but weak" algorithms, because it incorporates the effectiveness of compression.

However, this is still not correct, because the EUR fails to consider the possibility that the network may or may not be the bottleneck. For example, consider a compressor X that has a compressed output rate (i.e., *compressed_size / compression_time*) of 100mbit/s and that X achieves a CR of 0.50, or a gain of 2.0. Thus, the EUR would be (*100 * 2*), or 200Mbit/s. However, if the *available* bandwidth is only 1mbit/s, the maximum amount of information that can be moved using X is (1mbit/s * 2.0), or 2mbit/s.

On the other hand, suppose there is a compressor Y that achieves a CR of 0.33 (a gain of 3.0) on the same input. However, compressor Y's maximum compressed output rate is only 900kbit/s (0.9mbit/s) – more than 100 times slower than X. Based on these numbers, X has an EUR of 200mbit/s, while Y's EUR is only 2.7mbit/s. However, if the network bottleneck of 1mbit/s is applied, Method X's actual rate is only 2mbit/s, while Y's is *still* 2.7mbit/s.

Intuitively, we can say that for any given Method, if the I/O channel is *faster than the compressor*, the compressor's output rate times the compression gain is important, because the compression gain effectively enables the compressor to potentially meet or exceed the output rate of the channel. However, if the I/O channel is the bottleneck for a given Method, then the only property of the Method that matters is the compression gain. This matches our observations that "slow but strong" compressors can improve efficiency for slow networks, while "fast but weak" algorithms are the only ones likely to succeed for fast networks.

This provides the answer we need. By using the compressed output rate, the compression gain, and the EABW (a statistic we have already attempted to harden from the effects of buffering), we can calculate and compare the performance of various Methods in the same opportunities, without relying on the *operation_time* statistic that is unreliable for Datacomp at the level of individual events. Thus, we calculate the Estimated Effective Output Rate (EEOR) as:

$$EEOR = min(EABW, compressed\_output\_rate) * compression\_gain$$

If the EABW is less than the compressed output rate of the Method, then only the EABW and the gain matter. On the other hand, if the compressed output rate is less than the EABW, then the output rate and the gain matter. The EEOR is the key statistic for `time_decider()`, since it is an estimate of the effective output rate (EOR) of a particular event. By maximizing the EOR (as described above in Section 5 in the discussion of Comptool), Datacomp can minimize the total run time of the operation. In the next section, I describe how Datacomp uses the previous tools to decide which method will be most effective for the given opportunity.

6.5.3.3 History Database

The history database for TD contains the mean EEOR for every type of Event – every combination of Method and Opportunity. A Method is defined by the unique characteristics of that compression technique, and (as described in Section 6.2) can be described as a tuple of algorithm, strength, chunk size, and thread count. Due to the quantization described in Section 6.5.3.1, individual Environments can also be described by the tuple *(data type, available bandwidth, CPU load, CPU frequency)*. This means that every Event can also be identified by a tuple – the unique combination of specific Method and Opportunity tuples.[61]

In order to track performance and predict Method Results, TD builds and uses a persistent, in-memory database of mean EEOR results for every Event type. Given a tuple identifying the current Opportunity, TD scans the Results for the associated Event tuples in order to find the Method with the best EEOR for the Opportunity. TD instructs Datacomp to use this Method. After it performs the compression, TD updates the database record with the new Results.

Given the information already discussed in this section, the history database is fairly straightforward to explain. The database is implemented as a memory-mapped file, which is opened at start time and written to disk upon close so that it persists across individual Datacomp instances. In memory, the data is composed of "perfrows" – records composed of three 32-bit integers describing the mean EEOR of a particular event (shown in Figure 13).

Mean EEOR	Count	Sum

Figure 13. The time_decider() database record.

The layout of the data in memory is ordered by Opportunity first and Method second. This means that the "table" of Methods corresponding to a specific Opportunity can be directly accessed by offsetting into the database (achieved by multiplying the Opportunity tuple values by the size of an "Opportunity table" of perfrows). For example, there are 300 unique Methods in the current code (five algorithm families, three strengths, five chunk sizes and four thread levels). Each perfrow is 12 bytes, so each table of perfrows corresponding to a specific

[61] The database also includes separate parts for compression rate and decompression rate. The database mechanism for both is the same, but rather than being used for making choices, the decompression table is primarily intended for logging and testing.

Opportunity is 3,600 bytes. To find the start of the table corresponding to a specific Opportunity, for example, one defined by a CPU load of 1-33% (level 1), CPU frequency of 50-74% (level 2), an EABW of 20-200mbit/s (level 3), and a bytecount of 34-66 (level 2), TD offsets into the database by *(1 * 2 * 3 * 2) = 12 * 3,600 = 43,200 bytes* (where 3,600 bytes is the size of a single opportunity table). Then, finding the best EEOR for the Opportunity is a simple matter of scanning the 300 perfrows in the table and returning the Method tuple corresponding to the row with the highest EEOR. Since all of this data is stored in memory during use, database access and operations are extremely inexpensive; individual reads (before a compression operation) and writes (after an operation) take approximately 0.000001s each – much less time than bytecounting or other operations.[62]

The value of primary importance is the mean EEOR. However, to maintain the rolling average, the database also stores the count of Events (the number of times this record has been updated), and the sum of the EEOR values used to compute the mean. When the record is updated, sum and count are updated and used together to compute the mean EEOR. To ensure that the EEOR can change over time and also that the value *sum* does not overflow, the value *count* is never greater than 20; once 20 individual Events are incorporated into the record, future Events are only added after removing the equivalent of one mean EEOR from sum and decrementing the count.

Because of this learning approach (accumulating results using a rolling average), the database must be trained before it will provide useful advice. In real life a user might be content with having Datacomp default to a "safe choice" (such as no compression or zlib) if the database has not been fully trained. However, since one of the goals of this research was to explore the performance implications of a wide variety of choices, I built the TD database to treat every

[62] Incidentally, this suggests that the database could do quite a bit more work without becoming a burden.

method as potentially equally valuable and to intentionally try unused methods first – until all methods had been tried at least twice. This means that TD will skip directly over a profitable choice if there is an unused choice in the appropriate opportunity table. This can lead to performance issues during training, e.g., using xz on a gigabit network. However, it is effective in training the database comprehensively for testing purposes.

As previously described, some AC systems are susceptible to a “death spiral.” This can happen when compression is increased when performance decreases. If compression degrades performance in this situation, the compression level will simply be ratcheted up, which only exacerbates the problem. TD does not suffer from this problem, because it will consistently choose the Method with the best performance figures within the applicable Opportunity table. If one Method degrades performance in that context, and the degradation is sustained, the mean value will eventually be lower than some other Method (20 events is not long in Datacomp time), and at that point the alternative Method will be chosen. If all actual compressors perform poorly, eventually compression will fall back to “null compression.”

The worst-case scenario for the TD database is one where some kind of realistic and sufficiently common type of data exists but which has radically different Event Results than the other Data in the given Opportunity, and which appear only often enough to “poison” the database. If this happens, performances would increase when the “good” data was compressed, but would degrade when the “bad” data was compressed. In extreme circumstances it is conceivable that the database would “thrash” – approaching some mean for a certain amount of time under the “good” data but then moving back towards the other mean when processing the “bad” data, or that the performances would just settle around some mediocre value. However, if the database *was allowed to learn during this Scenario* and the performance was degraded sufficiently, the database would ultimately be led to use a different Method for the Opportunity, including potentially null compression, for which the performance does not change based on data characteristics.

Another potential pitfall faced by the TD database (that does not require pathological data) is that a Method for an Opportunity could become artificially "devalued" – its average EEOR estimate stuck in a minima – and because of this would not have a chance to remediate itself. For example, suppose that the results for some Method A beat the results for Method B, but that the results *should* be the other way around – that Method B is actually superior to Method A but simply performed poorly when tested. Once Method B's mean EEOR is lower than Method A's, it will normally not be tried again unless Method A's results dip below Method B's. In other words, in the current Datacomp, "second place is just first place loser."

To help make sure that over time the values settle on accurate results (without unduly taxing the database with speculative choices), the database's updating function will randomly "handicap" Methods through boosting the EEOR value by 10% to see if they will be selected on the next sweep. Rather than updating the sum or count of the perfrow, the mechanism merely bumps up the mean EEOR field in the perfrow, but leaves the sum and count intact. When the table is scanned, it will find the new value to be the best if it was less than 10% behind the "real" leader. If that happens, TD will use the "boosted" Method. However, once the Method is tried, the result will be added to the historic sum and the count will be incremented; in other words, the boost is ephemeral but the mean EEOC will be recomputed with the new results.[63]

[63] This mechanism only operates if every method in the table has been tried the minimum number of times (currently twice) and does not happen for every database write.

6.5.3.4 Discussion

I see two main weaknesses to the TD model, although neither is fatal. The first is that there is no guarantee that the current set of Opportunity tuples (i.e., the monitored values and the quantization mechanisms that reduce them) is ideal. It is almost certainly the case that the current design can be improved. For example, it could be that various levels used in the TD database are not useful, or that the current bytecounting approach is not powerful enough to improve efficiency enough. By changing the levels, adding additional Monitors, tweaking (or replacing) database mechanisms, I think that accuracy could be improved. In any case, doing this should be relatively straightforward.

The second problem is more difficult. The power of the current design is its fine granularity, which comes from the large number of properties that are considered in making decisions. Unfortunately, because of this granularity, training can be costly, requiring multiple days to make sure that every Event type is trained. As previously mentioned, reducing the granularity of the database or having sensible fallbacks to known good methods rather than exhaustively trying every Method could make training "on the fly" practical, but I did not explore this.

Another possibility is that a large amount of the training data or "advice" encoded in the database may be broadly portable. One reason to think this might be the case is that the specific mean EEOR values stored in the database are not as important as the ranking of the methods for a given opportunity. If the performance properties are generally consistent across various types of hardware, it could be that a long training period or a starting set of sensible "bootstrap" values could be a one-time investment rather than a process that every device would be subjected to. These are issues for future work.

6.6 Example Application: dzip

Dzip is a simple utility that attempts to achieve maximum compression of any input without requiring any assistance from the user. Rather than using one or more manually selected and hand-tuned algorithms, dzip uses Datacomp to perform the compression, configured to prioritize saving space over time or energy. The end result is a file compressed using a greedy strategy which chooses the best-compressing method at every opportunity. In this section, I will discuss dzip's basic interface, operation and use of Datacomp.

6.6.1 Interface

Dzip emulates (part of) the basic interface of gzip and other similar compressors, in that it reads from a standard input or a file and writes data to standard output. Dzip includes a decompression mode, where it reads dzip-compressed data from standard input or a file and writes the uncompressed data to standard output. In this way, dzip can be used as part of a "pipeline" in the same way that compressors such as gzip are used.

For example, to compress a given file, one can execute the command:

```
$ dzip foo.txt > foo.txt.dz
```

To compress a stream (e.g., a FIFO) on the fly and output it to another stream:

```
$ cat /dev/fifo1 | dzip | /dev/fifo2
```

By default, dzip uses Datacomp to greedily optimize the compression to save space. However, because dzip relies on Datacomp, it can also optimize for time (or energy). For example, since dzip reads from standard input and writes to standard output, it can be used in an attempt to optimize a pipeline for time:

```
$ cat /dev/fifo1 | dzip --time | /dev/fifo2
```

The remaining details of dzip's interface relate to testing. For example, dzip allows users to request specific compressor types, chunk sizes, numbers of threads, strength levels and so on. When these command line parameters are used, dzip instructs Datacomp to use the "manual" mode as defined by the given options (and as described in Section 6.5.1). Datacomp can also log the choices made and various performance timings taken during operation. These features are useful for AC researchers interested in testing Datacomp's performance in specific conditions, such as comparing dzip's performance using a given low-level method provided by zlib versus a standard compression utility such as gzip.

6.6.2 Dzip Internals

Dzip is essentially a file-based interface wrapping Datacomp. Dzip provides the command line interface (e.g., reading and decoding parameters) and file management (opening and reading input and opening output). While the Datacomp Stream Format (DCSF) includes a header (DCSH), this header only defines the properties of the given Datacomp Frame. Thus, dzip defines a header for dzipped files. This header, which is written to new output files (and verified when decompressing dzip data) includes a magic number identifying the data as being "dzip format," followed by the length of the uncompressed data (which may or may not be used, depending on dzip's mode). The header does not include any other integrity-checking mechanisms (such as checksums), although it is of course encapsulated in a DCF as well.

Once started, dzip instantiates the Datacomp File Descriptor (DCFD) by passing a plain file descriptor and a priority to libdatacomp. The priority is usually "space," but can be "time" or a specific manual mode. After the new DCFD is returned by libdatacomp, dzip writes data to the DCFD via the function `dcwrite()`. Internally, Datacomp determines which compression methods to use based on the model used (described in Section 6.5) and writes the compressed data to the underlying file descriptor.

Datacomp also provides decompression; in this case, dzip first opens the Datacomp-compressed file for reading and passes this file descriptor to libdatacomp, creating the DCFD for reading. This DCFD is read using `dcread()`, which returns the requested number of decompressed bytes. No priority is necessary when instantiating a DCFD for decompression, since Datacomp will use whatever method is required as defined by the DCFH (see Section 6.3.2).

6.7 Example Application: drcp

Drcp (i.e., Datacomp Remote Copy) is a simple network transmission utility that uses Datacomp to improve the efficiency of the channel with respect to some resource. Typically, drcp attempts to achieve maximum compression throughput of any input without requiring any assistance from the user. Rather than using one or more manually selected and hand-tuned algorithms, Datacomp selects the strategy and performs the compression. The end goal is a network transfer tool that over time achieves greater efficiency than could be achieved through using standard static compression mechanisms. In this section, I will discuss drcp's basic interface, operation and use of Datacomp.

6.7.1 Interface

Drcp's interface is extremely similar to that of dzip, with the exception that drcp must specify options necessary for making network connections. Drcp also includes TCP server code; since drcp is a network transfer tool with compression performed on the client side, a drcp "server" must be started on the remote host. Finally, drcp can transfer a list of files one at a time, while dzip processes only one file at a time (this is to reduce startup/shutdown costs during testing).

Otherwise, most options are similar, and include specifying the manual compression method or the dynamic decider space or time. Like dzip, drcp includes a large number of options intended for testing and debugging, including switches to enable the writing of the `time_decider()` database to disk or SQL, to print a log of the compression choices used, disable model learning, and so on.

For example, executing the following command will start a drcp server on a host:

```
$ ./drcp --listen 10000
```

Sending a file using drcp in client mode is also straightforward:

```
$ ./drcp --host 10.10.10.10 --port 10000 --file /tmp/input --time --dest
/tmp/output
```

This command would transfer the file `/tmp/input` to host 10.10.10.l0, creating the file `/tmp/output` on the remote host. Datacomp would attempt to maximize throughput by using the "time" decider.

6.7.2 Drcp Internals

The internals of drcp are very similar to those of dzip. Instead of passing a regular file descriptor to `make_dcfd()`, drcp makes a socket connection to the server first, and then passes the socket file descriptor to `make_dcfd()` along with the resource priority (e.g., time). Datacomp can identify the difference between regular files and sockets, and uses the appropriate underlying system call for reading and writing. Like dzip, drcp can use any model included in

Datacomp. While it is intended to be used with the time decider, one could use the space decider in an attempt to minimize the total number of bytes sent over the network.

Unlike dzip, compression and decompression are typically connected in drcp. Drcp, (through Datacomp) automatically decompresses the compressed input on the fly as it is read from the DCFD. As a result, the output file on the remote server is uncompressed data. This was done to mimic the behavior of file transfer tools and to enable testing of the effect of decompression time on overall throughput. It also means that dzip and drcp have different "application headers."

While dzip's header is prepended to the compressed file so that it can be decompressed later, drcp's header is the first payload data in the first DCF. It consists of three 32-bit integers and a variable-length character array. The first integer is the length of the filename, which is needed to compute the size of the rest of the header. The second value is the length of the file, and third value is for special flags.[64] Finally, the destination filename is the remainder of the header. Once this header is read into memory, the server, through Datacomp, reads and decompresses the appropriate subsequent data frames to the specified output filename.

6.8 Discussion

Dzip and drcp are demonstration applications meant to show the power and flexibility of Datacomp, described in this section. Datacomp's power is in its ability to independently monitor a wide range of properties that affect compression outcomes, to quantize those properties into manageable and discrete levels, and to use arbitrary decision mechanisms – ranging from the simple to the complex – in the attempt to improve efficiency using compression.

[64] Currently, this area is only used to indicate whether there will be additional files following the current file. If there are not, the socket is closed following the last file. If there are, a new drcp header is read.

Datacomp's Methods are diverse, and it is straightforward to add new compression Methods. Its mechanisms do not require any third-party systems or special support, they automatically perform compression and decompression in parallel, and they allow for easy integration into existing code.

For example, to incorporate Datacomp into a web browser and server combination, the socket code used to communicate between the web server and client would be replaced with calls to their Datacomp equivalents, `dcsend()` and `dcrecv()`. All traffic over this channel would be adaptively compressed based on whatever priority was specified when the Datacomp File Descriptor was instantiated. For other, non-Datacomp files or sockets, the software would simply use legacy system calls.

The set of Monitors provided by Datacomp covers the most significant elements defining the compression environment – the CPU load, CPU frequency and bandwidth, and the bytecounting mechanism described here provides a simple and efficient means for generally assessing the compressibility of data. Datacomp's Models include simply applying a pre-selected compression Method, using a greedy approximation to maximize compression ratio, and a complex decision mechanism based on a continually updated history database. Importantly, almost all of these components are specifically designed to be modifiable or replaceable so that researchers can examine and explore various approaches to Adaptive Compression.

For example, decision Models have almost arbitrary latitude in terms of how they can make decisions or what software they can execute in the process of making those choices. The API is straightforward; Datacomp provides the Model with a buffer and length, and also provides the standard monitored environment variables in an accessible structure. The Model takes this information and generates whatever other input it needs (such as by bytecounting the data). It can then consult databases, external sources, or literally anything it can programmatically access. (In this way, Datacomp *could* use third-party host support if it was desired). The Model then calls the compression functions included in Datacomp, and returns to the caller the number of bytes that were consumed. In essence, if it can be programmed, it can be included in a Datacomp model.

Datacomp also includes two example applications – one using a regular file interface to implement a file compressor, and the other using a TCP socket to implement compressed network communication. While each application was designed with a specific Model in mind ("space" for the file compressor and "time/energy" for the network copy tool), either application can use any Datacomp Model because the Model and all the underlying mechanisms are abstracted away within Datacomp. As a result, these example applications should likewise be useful for future research as well.

Datacomp is a proof of concept and a work in progress. While it is successful (as will be detailed in the Evaluation), there is considerable room for improvement.[65] Nevertheless, Datacomp has shown that a locally-independent Adaptive Compression system relying on a minimum of hard-coded knowledge or design choices can meaningfully improve the efficiency of communication in real-world scenarios.

7 WORKLOAD SELECTION

7.1 Introduction

This research has two primary goals. The first goal is to show whether AC is a potentially valuable technique with the promise of saving resources in meaningful, real-world scenarios.

[65] For information on using, extending, or improving Datacomp or Comptool, see Appendix A.

The second goal is to demonstrate, through the use of Datacomp, that Local Adaptive Compression is practical and capable of making effective decisions to improve efficiency within similarly meaningful and realistic scenarios. Accomplishing these goals requires the selection and re-creation of these scenarios, in addition to the careful selection and execution of experiments using Comptool and Datacomp.

7.2 Scenario Design

It is difficult, if not impossible, to design a "typical" workload for AC. First, there is no general consensus of what "typical" data for compression is. Workloads such as the Calgary and Canterbury Corpora,[66] while potentially still useful for compressor design, are too small and limited for evaluating a compression system such as Datacomp. In addition to being old, they simply were not designed for this purpose. For example, the Canterbury Corpus was created over 15 years ago and contains approximately 2.7 megabytes in 11 files. Eight of the files are plain text of some kind, such as source code (two files) natural language (four files), and files using a markup mechanism (a GNU manual page and a page of HTML). Only three of the eleven files are binary, including a Sparc executable, a CCITT fax bitmap, and an Excel spreadsheet. Only the last two are particularly difficult to compress.

A fundamental problem with the corpus is that it simply is not large enough to test an AC system. A corpus for AC testing must be large enough so that the adaptation mechanism cannot simply be "trained to the test." In other words, if there is an ideal strategy for compressing a particular file in a specific environment, the adaptation mechanism might find it, especially if the same file is repeatedly tested. At that point, the usefulness of the file as a workload decreases significantly. A real-world AC system must be able to perform well on data it has never seen

[66] http://corpus.canterbury.ac.nz/descriptions/

before, which presumes that there is enough data to thoroughly test it without repetition. Since the Canterbury Corpus has only 11 files, it is quite possible that an AC system could simply "learn" the best strategies for that data.[67]

Even if the corpus was large enough to support many unique experiments required for an evaluation of AC and Datacomp, it is not diverse enough in general (i.e., it does not have multiple examples of each type of file) or similar enough to typical user data to be a convincing workload for demonstrating the general usefulness of AC. An AC system that could be worthwhile for general use must be able to cope with a heterogeneous mixture of compressible and uncompressible data, and the Canterbury Corpus is almost entirely compressible.

Even though we can say that the Canterbury Corpus is not a very useful corpus for this work, it is not easy to identify what would be. Designing a modern corpus for AC is a project in itself, and would likely need to be continually updated to reflect current usage patterns and applications. In the real world, data characteristics vary quite widely based on the application and there are too many applications in use to make general assumptions.

Due to this diversity, it might be impossible to create a "one size fits all" compression test. However, this project focuses on the general computing environment, that is, the kinds of things that end users do on a day-to-day basis and the data they use in those tasks. This includes things like desktop files, email, applications, and web traffic.

It is difficult to say with any authority what proportion of these data types "typical" people use. What is "typical" no doubt varies from person to person, and can change over days, months, and years. However, we can collect sets of data from common tasks without making assumptions about exactly what an individual's use looks like. Then, we can perform tests on these classes of data and let the results speak for themselves.

[67] Additionally, there is not enough data to reasonably separate the files into training and testing sets.

The important characteristics of the data for this project are that:

The data is realistic. The input needs to be composed primarily of actual real-world data, including user-generated files from desktop applications, multimedia such as MP3s and compressed images, binary applications themselves, email, and web traffic – collected in a way that mimics or emulates realistic use.

There is enough data. Compressing the same files repeatedly in the same way is not sufficient to evaluate an AC system. First, with a limited set of input, it is not possible to have confidence that the data is generally representative of its class, because individual files could have characteristics that make them artificially easy or pathologically hard without the corpora designer's knowledge. For example, there is only one executable file in the Canterbury Corpus. How can we know that all executable files have similar compression characteristics?

Beyond this basic problem, the main reason we need a large workload is that AC often prioritizes time or energy over compression ratio. Time and energy Results are not fully determined by the input and Method used; they are affected by the Environment, which is naturally variable. Even if the Environment could be described at a high level ("80% free CPU at 100% frequency using a 10-megabit network"), the resulting performance would not be absolutely consistent, but would fall within some probability distribution. The stochastic nature of this problem requires multiple repetitions of experiments to ensure statistical confidence. But if those multiple repetitions are performed on the same limited number of files, this simply creates a statistically confident measurement of a very limited test. To have confidence in our experiments on a particular class of data, we need enough unique data to run multiple repetitions of convincingly-similar-but-different experiments within that class.

The compressibility of data is representative and its weight is proportional to its importance. It is important that not all data is compressible or uncompressible, but that the data's compressibility reflects the real-world compressibility of the class of data it is meant to be a sample from. Some classes of data, such as text, are generally compressible. But other classes of data, such as web traffic, are composed of both compressible and uncompressible data. In these situations, it is important that the proportion of compressible to uncompressible data[68] in the class is representative of the original data. Additionally, while compressibility should be treated as a ratio, classes should be collected by size, not by numbers of files. A serious mistake would be to collect a set *number* of different types of files (e.g., 100 files each of the classes "binaries," "text", "images," and "audio"), without regard to the size of the files or their proportion of the population in reality.

For example, it would be an obvious mistake if I attempted to make a class of "user data" where I collected one file for each unique file extension (e.g., .txt, .html, .jpeg). First, this approach disregards the average size of each file type; I might pick a 15-byte text file at random, when the "average" file of a file with the extension .txt is actually 37K. Secondly, selecting by extension completely ignores the proportion of storage space devoted to each kind of file. Modern users tend to have a large collection of media in terms of personal pictures and MP3s; these files are larger than most productivity documents or other human-generated data. So, in practice, these kinds of files occupy the bulk of the bytes in a collection of "user files". Seemingly practical choices can lead to data that is unrepresentative of the real world.

For example, suppose that in order to limit input data to a particular maximum size (e.g., to keep experiments of manageable length), workload files were truncated after a certain size,

[68] Of course, in reality data is not simply "compressible" or "uncompressible"; if data is compressible, it's compressibility is on an effectively continuous scale.

say, one megabyte. Small files would be intact, while large files would still be at least one megabyte. However, this artificially inflates the proportion of data in smaller files by making it appear from a statistical perspective that no files on the system are larger than one megabyte. Smaller files, which are less likely to already be compressed and thus are more amenable to compression, would be artificially inflated in importance, and thus this "practical choice" (made to limit test run times) would inflate the value of compression relative to how it would perform in the real world.

The data should include a wide variety of types and contents. Because it is difficult to create a single representative set of input, it is important to include a wide variety of different types of data that may be of interest to others. Using only documents, or only web data (or even only data from a particular site) can result in the results being less significant in a practical sense, since other users and researchers may not be interested in the limited range of data used. Thus, it is important to include a wide variety of data with different compressibility characteristics.

The data includes predictable and reasonably well-understood input. At the same time, while it is important to include a wide variety of data, if it is impossible to characterize the data in a meaningful way, it will similarly be difficult for users or researchers to apply to their own work. Thus, each class of data must have reasonably understandable characteristics or properties that will be meaningful to others.

The data includes best and worst case examples. Since it is not possible to include data of every possible type, it is also important to include best and worst case examples so that the performance of the system on extreme examples of input can be understood. This is not only interesting, but it can help to bound expectations, since in practice no compressor can do better than on the easiest possible input, and no worse than on the most difficult input.

Finally, this project is not a compressor evaluation exercise. We are not simply testing the efficacy of various compressors on different classes of input data. Instead, we are testing the effect of compression in a broad range of different Opportunities defined in terms of Data and Environment. This requires methods to set and control various environmental parameters, such as the network bandwidth, CPU load, and frequency. This is a much more straightforward process than developing a compression workload, although there are meaningful challenges especially to accurate bandwidth limitation.

7.3 Data

In this section I will discuss in-depth the data collection procedures and rationales I used to create and sample the different classes of data for use in my experiments. These procedures were designed to fulfill the criteria described in Section 7.2. The three broad classes of data used for these experiments are "Controlled Data", "Uncontrolled Data", and "Extrema." Where appropriate, I will also describe interesting design choices or instructive problems that I encountered.

7.3.1 Controlled Data

"Controlled Data" is data that I collected using a specific, structured method. While I cannot make all of my experimental workloads public for privacy reasons, interested parties should be able to recreate my collection procedures to collect data with similar properties. While Controlled Data is somewhat artificial in this sense, it is also the most repeatable and accessible to others (of the real world data used). By collecting "controlled data" for common tasks and popular websites, I created workloads that are both consistent and convincingly representative of real-world user tasks.

7.3.1.1 Web – Facebook, YouTube and Wikipedia

Web use is a major component of the workload of typical user computers. While it is impossible to know what the "typical user" does while on the web, we can identify the most popular websites. Facebook, YouTube, and Wikipedia are all among the top seven websites worldwide [99] as of this writing. In fact, if one removes the search engines Google, Yahoo!, and Baidu, the #1, #2 and #4 web sites are Facebook, YouTube, and Wikipedia, respectively (Amazon.com would be #3). These sites are thus an excellent example of "typical web surfing" by global volume.

These sites also have useful properties from an experimental perspective. First, they have different compressibility characteristics. Intuitively, Wikipedia[69] is likely to be the most compressible of these three, being mostly text, with some Javascript (a language interpreted within browsers to add functionality to websites), Cascading Style Sheets, and a small number of images (in comparison to other sites). Facebook is likely to be less compressible than Wikipedia because it includes many more graphics. However, it also contains a large amount of text and code, and many of the images are reused, such as users' "profile pictures." Finally, YouTube is likely to be the least compressible of these three. While it also includes text and code, the vast majority of data served by YouTube is compressed video.

Second, each of these sites can (as of this writing) be viewed in a fairly deterministic manner that is arguably still representative of the general site experience. Wikipedia has a long list of "Featured Articles" that are displayed on the homepage and which represent what are understood to be the highest quality, most editorially mature, and most generally interesting articles on Wikipedia. Thus, they represent the experience of reading high quality Wikipedia information. By loading a list of these featured articles in my browser one at a time, waiting for

[69] http://en.wikipedia.org/

it to finish loading, and repeating until I downloaded a set number of bytes, I downloaded a large cache of high-quality Wikipedia documents.[70]

Facebook has a unified interface that is consistent for all users browsing the site. "Status updates" posted by users are shown in roughly chronological order in the main pane of the site, with notifications and advertising on the right. Facebook uses Javascript running in the browser to load more content once the user has paged to the bottom of the currently downloaded information. Thus, to download a long stream of typical Facebook data, I simply scrolled to the bottom of the page, waited for the content to load, and repeated the process until I gathered a preset number of bytes.

Like Facebook and Wikipedia, YouTube has mechanisms that enable browsing in a fairly mechanistic way while still getting the typical experience. For YouTube, I simply watched the most popular videos[71] in the top categories until I downloaded a preset amount of data. YouTube hides most comments and other content unless users specifically want to view it. To simulate *my* typical YouTube experience, I skipped the comments; when a video finished playing, I selected another video and continued watching.

To capture this data, I browsed the sites using Firefox while capturing data with the popular "sniffer" Wireshark [100]. Wireshark, which uses tcpdump's [91] library libpcap, includes the ability to stop capturing data after a certain number of bytes have been downloaded or a time limit has been reached. For my purposes, a space limit was more appropriate, since I wanted a realistic sample of the data and I wanted to ensure that I had enough data for many individual experiments, as opposed to being focused on network or user-centric characteristics

[70] Wikipedia also includes a "random article" tool, but since many Wikipedia articles are "stubs" – short placeholder articles without much detail, my judgment was that choosing articles randomly was not representative of my experience reading Wikipedia and thus I used the "featured article" lists.
[71] Similar to Wikipedia, I used the most popular videos rather than selecting random videos because much of the content on YouTube is rarely viewed and I wanted the results to be more "typical" than random.

that might depend more on time (e.g., “How many bytes does the typical Facebook user download in a single session?”). Capturing data with Wireshark created “pcap” files from which I extracted the server-to-client payloads using the tool chaosreader [93]. This resulted in one file containing the data in each observed HTTP session. The contents of these files were not identified or filtered in any way, so the set of files contained a realistic mix of HTML, markup, source code, and multimedia data.

Normally, a browser will inform web servers that it is capable of decompressing DEFLATE data, and will of course use encrypted data if it is downloaded over HTTPS [101]. However, it is important for this project that the captured data not be encrypted or already compressed, as this would obviously hinder any compressibility investigation. Fortunately, Firefox has a fine-grained internal configuration mechanism, “about:config” [102], that allowed me to disable those features directly.[72]

7.3.1.2 Files – Binaries and Email

I also wanted to test some file data that could reasonably be expected to have different characteristics than web data, would be representative of tasks that users perform, but that was nevertheless distinct and limited in some way. I chose to collect ELF binary applications [103] as one class of data and a large collection of emails (including attachments in realistic proportion) as another.

I acquired the data for the “binaries” class by copying the entire set of installed programs from the /usr/bin directory of my Ubuntu Linux 12.04 laptop. This totaled approximately 425 megabytes. For email, I copied my two most heavily used mailboxes (one personal, one research

[72] When collecting Facebook data, I had to change my account security settings to allow non-encrypted access. I would not be surprised if this “feature” is removed in the future, however other researchers should be able to use some kind of proxy-based solution to extract the unencrypted data if they wish to create similar data.

related) totaling 488 megabytes into one directory. The mail was then converted from mbox [104] to Maildir format [105], so that each message (many of which include attachments) would be contained in a separate file. The individual files in each class were kept separate (i.e., not concatenated) in order to preserve their sizes and distinct characteristics.

These two types of files were selected because they represent common non-web, non-productivity files read from disk or transferred over the network. It is reasonable to assume that they are fairly representative in that the binaries were copied in whole from the system and the email is heavily used and contains a mixture of both long and short messages with and without attachments.

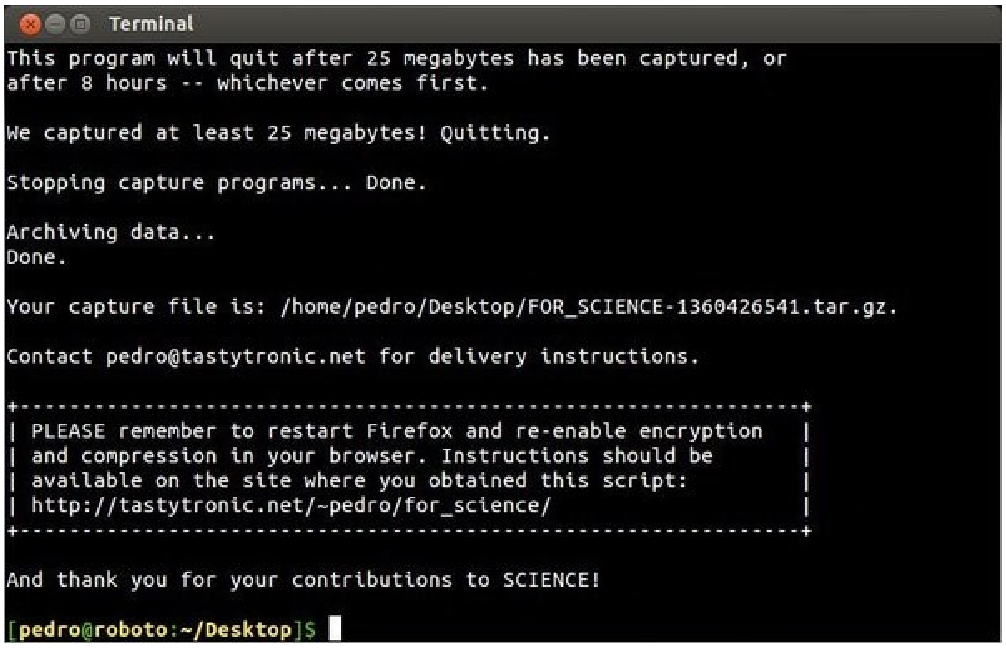
7.3.2 Uncontrolled Data

Because we know that users do not browse only Facebook, Wikipedia and YouTube, and do more with their computers than load binaries or read email, I felt that it was important to include unconstrained web browsing and user data files. Additionally, since people use their computers in different ways, it is important for this data to come from a variety of sources – not just my own computer. Collecting data of this type is challenging, however, because of security and privacy risks.

7.3.2.1 Uncontrolled Web

To collect uncontrolled web data, I wrote a script to collect user web traffic in 25 megabyte chunks and created a web page [106] with detailed instructions for downloading and using the script (including disabling encryption and compression in Firefox). The script uses tcpdump [91] directly, since it is installed by default on current Apple OS X systems and is also commonly installed on Linux desktops. After I received the capture files from my volunteers, I used chaosreader to extract the data as with the Controlled Web data described in Section 7.3.1.1.

The web site goes into detail about the security ramifications of assisting with this effort, and includes explicit instructions for avoiding including personal information (by using a “Private Session” in Firefox) and for re-enabling both encryption and compression in their browser. The instructions also indicate that the users should not use websites with any sensitive personal information, including banking, e-commerce, or Facebook. While this somewhat limits the users’ behavior, I felt that these limits were acceptable, since collecting data for sensitive tasks (e.g., e-commerce) is not worth the privacy liability in my opinion, and I had already collected Facebook data myself.



```
This program will quit after 25 megabytes has been captured, or
after 8 hours -- whichever comes first.

We captured at least 25 megabytes! Quitting.

Stopping capture programs... Done.

Archiving data...
Done.

Your capture file is: /home/pedro/Desktop/FOR_SCIENCE-1360426541.tar.gz.

Contact pedro@tastytronic.net for delivery instructions.

+----------------------------------------------------------------+
| PLEASE remember to restart Firefox and re-enable encryption    |
| and compression in your browser. Instructions should be        |
| available on the site where you obtained this script:          |
| http://tastytronic.net/~pedro/for_science/                      |
+----------------------------------------------------------------+

And thank you for your contributions to SCIENCE!

[pedro@roboto:~/Desktop]$ 
```

Figure 14. The end of the for_science.sh capture script.

In total, five volunteers and myself contributed uncontrolled web traces, which, after extraction resulted in 317 megabytes of uncompressed server-to-client web data, which I combined into one pool. In contrast to the Controlled Data, I generally do not know anything about what the traces are composed of other than that they shouldn't include economic or personal traffic. In fact, I informed the volunteers that I would not inspect the data in any way unless it was specifically necessary for debugging. However, I do know from informal conversations with volunteers that the traffic includes streaming media, news sites, and so on.

7.3.2.2 Uncontrolled Files

Uncontrolled files are meant to represent the kind of data that users use on a regular basis, including productivity files, pictures, music, and other documents. While collecting even limited web traces is a privacy and security risk for users, submitting caches of their personal files is even more of a risk. However, I felt it was essential to investigate how AC and Datacomp might work on these types of data.

To model this data class, I collected files from the personal computers of myself and three close friends. All the volunteers are computer-savvy students or professionals who use their computers regularly. Each volunteer had large amounts of productivity documents, music, and pictures. For the collection process, I asked the volunteers to indicate which folders in their home directories I should *not* index for privacy reasons. Then, I randomly selected files from the remaining directories, up to some number of bytes. File names were discarded and replaced with numbers, although the file extensions were retained for compressibility analysis. These files were then combined into one large pool of user data totaling two gigabytes.

There are three specific exceptions to this otherwise uncontrolled capture process. I have a large number of compressed archives on my laptop but which I do not access on a regular basis. To avoid having these files skew the distribution, I limited the files from my laptop to those files less than 60 megabytes. No such limitation was made for the other users. However, for one user, I explicitly ignored a single 600 megabyte movie file, because it was a clear outlier and dominated the input in terms of bytes.[73] For another user, I ignored a folder that had been created as part of a music library backup. The folder had simply been left behind and was not in use, but duplicated many of the files already in the capture. In all cases, I ignored zero-length files (empty files with names but no data).

It is worth noting that, unlike the web data, which was actually captured in the context of it being used, the user files (both controlled and uncontrolled) do not include any usage information (e.g., frequency of use, modification dates or access dates). For example, while I put every binary in /usr/bin into the pool, most of those applications are not used on a regular basis. Similarly, while the user data consists mostly of images and music, frequency of use statistics were not recorded. Capturing this kind of data, perhaps by reading modification times or tracking usage over a period of time could be useful in the future, but was not a primary goal for this project.

7.3.3 Extrema

While I have included real-world data from users, and tried to include realistic data classes that I expected to be both compressible (e.g., binaries, email, Wikipedia) and essentially uncompressible (YouTube), I can't know at the outset whether these classes will span the full breadth of compressibility. Testing best and worst case data classes – data that is maximally easy or difficult to compress – is useful in a number of ways.

First, it can show us the best and worst possible performance of our Methods. Knowing the upper bound of performance we can deduce whether *any* compression Method could possibly be useful for a given opportunity. For example, if we know from past experience that no Method

[73] The movie file is also arguably similar to other uncompressible data such as images and music, which are already included in the uncontrolled user files, and to the streaming video captured as part of the YouTube controlled data class.

is able to improve efficiency for the *easiest possible workload* for a particular opportunity (such as the easiest workload possible in an environment with an extremely fast network), then other properties may not be necessary to obtain from Monitors, such as the CPU load or general compressibility of input data.

Worst-case scenario data can also help us by indicating a limit to the maximum cost for compression which could be useful for choice selection, particularly when making choices based on partial information. If the worst-case cost is acceptable for a particular transaction, then a "gamble" on a compression strategy could be worthwhile. But even without betting on compression choices, knowing the worst-case scenario can enable us to tune our Models more accurately. It can identify how Methods behave in extreme circumstances and can indicate how hard to compress a particular input truly is; if the performance on a given data is close to the worst case, then we can better categorize its results in the Model.

Creating these classes of data is trivial. To represent the easiest case, I created a number of "zero files" – files with length but which are simply full of null bytes. For the pathologically hard case, I created files using data from `/dev/urandom` – pseudorandom data. As there is ostensibly no structure in the data whatsoever, it should be impossible to compress.

7.4 Environments

To simulate a variety of Opportunities and Scenarios, along with varied Data types we also need a selection of different Environments. While any environmental property that *could* impact the performance of compression or compressed communication is of interest, there are three characteristics that we know are extremely significant: communication channel bandwidth, the available computational resources, and the frequency of the CPU. Thus, we need to be able to control these properties in our experimental environments.

7.4.1 Communication Bandwidth

7.4.1.1 Background

Controlling communication bandwidth for AC experimentation is critical. While computational resources are local constraints (and may be managed by the system), communication bandwidth is often not configurable by the local system. Furthermore, when AC is being used to improve communication efficiency it is precisely bandwidth utilization that compression is trying to optimize. Methods that are inappropriate for certain bandwidths could be the ideal choice for a different bandwidth. Thus, when performing AC experiments, some flexible method to limit the bandwidth of the communication channel is required.

Practically speaking, the method used to limit bandwidth for experimental purposes depends on the channel type being simulated. Network channels can be limited by traffic shapers, of which there are many, including Linux's tc [107], wondershaper [108], BSD's dummynet [109] (also available on Linux through the ipfw kernel module), or the Click Modular Router [110]. I use dummynet (i.e., ipfw) for live network links, which I found to be more accurate than tc for my purposes. In my tests, ipfw runs on the destination host. In each test, a command is sent across the network to configure the rate limiter, and then the test is executed.

File channels can be limited through the use of an application that supports rate limiting and functions like a Unix pipe; several such applications exist, including `pv` [111], `throttle`[74] and `cstream` [96]. While these tools work well for certain workloads, they did not meet my needs for reasons specified in Section 5.3.3. As a result, for Comptool I used a hybrid mechanism based on an actual write, but rate-limited based on a timing calculation. For dzip tests I did not limit the output rate.

[74] Throttle was maintained by James Klicman between (at least) 2003-2005, although the original homepage appears defunct. Source for throttle can be found at ftp://ftp.freebsd.org/pub/FreeBSD/ports/distfiles/throttle-1.2.tar.gz.

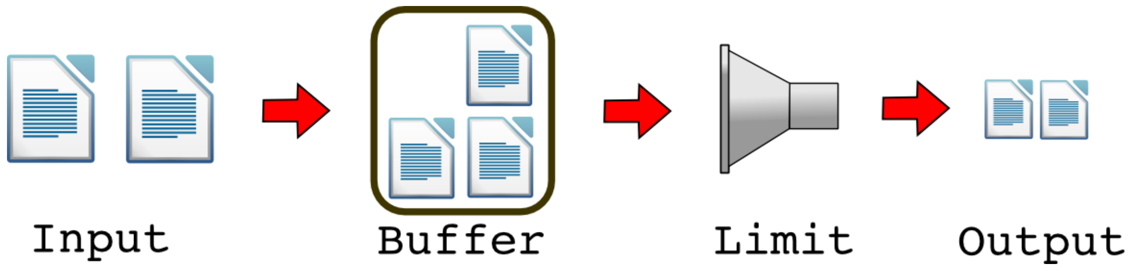


Figure 15. Buffering occurs before limiting -- limits will not impact the apparent send rate until the buffer is full.

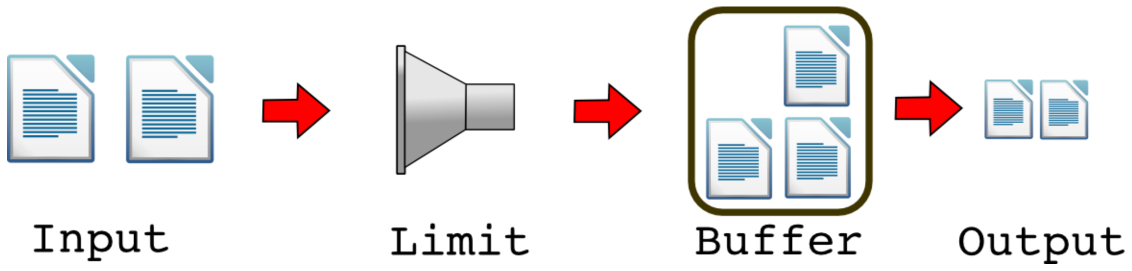


Figure 16. Limiting occurs before buffering and thus will impose limits regardless of the buffer architecture.

A related, but critical issue for experimenters is the interaction of buffering and rate limiting. While this is discussed in more detail in Section 5.3.3, I discuss the general issue here. If buffering occurs *before* limiting, as shown in Figure 15, then writes to the buffer may happen at the maximum user-to-kernel throughput. In this case, the effect of the limit will not be felt until the buffer is full and the limit becomes the effective bottleneck. This might never occur, giving the impression of an unlimited channel if the message is sufficiently small or enough time passes between send events while the buffer is being drained by a reader.

The alternative, placing the limit before the buffer (shown in Figure 16), enforces the write limit regardless of any buffering, but can make it difficult to accurately control throughput since buffering helps to smooth the impact of quantization due to scheduling and shared access to limited resources. This can be a challenging issue for experimenters to control, because the structural relationship between buffering and limiting is not always apparent when assembling a system for experimentation.

7.4.1.2 Levels

Given a sufficiently powerful limiting mechanism, networks can be configured in almost unlimited ways, controlling not only bandwidth, but also delay, jitter, and so on. However, for simplicity, I chose to limit the network based on bandwidth alone. Because additional levels required more rounds of testing, I chose to set limits at a few specific levels rather than an even spread of limits. These levels are meant to represent common use cases in the general computing environment where AC might be applied. Accordingly, in my experiments, I use five levels: gigabit networks and 100 megabit networks represent common LAN configurations; 6 megabits per second represents home cable or DSL connections; 1 megabit per second limits represent fast cellular networking; and 500 kilobits per second (64 kilobytes per second) represents slower cellular networking.

7.4.2 Computational Resources

7.4.2.1 Background

Because the success of any compression algorithm in improving the efficiency of communication depends on the run-time performance of the Method in the given Environment, system properties affecting execution (primarily the runtime or throughput) of the Method are critical to investigate. For experimentation and evaluation, a researcher must be able to set and hold the various properties in one or more states in order to control those variables. While there are many potential computational properties that could be controlled, we focus on two: the CPU load (or available cycles for computation) and the CPU frequency – the clock rate of the system.

The CPU load is affected most obviously by the computational cost of the operating system and of the processes running coterminous with the AC or experimental system in question. In essence, the load determines how much additional computation could be performed in a given amount of time. The CPU frequency is generally controlled by the operating system, increasing and decreasing the clock rate to match demand (as permitted by the hardware capabilities). Increasing the clock rate improves throughput because more operations can be completed in the same amount of time. However, this increase comes at a cost. For a given system, running at a higher frequency requires more energy, drawing from the battery at a higher rate and generating more heat, while the converse is true for running at lower frequencies.

Both the CPU load and frequency can affect the best compression Method for a given opportunity. For example, it is conceivable that some Method, A, might be profitable when the CPU load is below 50% but that when load is higher the computation cannot complete quickly enough. Similarly, it is possible that some Method, B, might be profitable at a high CPU frequency, but not at a lower frequency. While this is intuitively obvious, it is not clear how or when these factors matter. As a result, we need mechanisms to control these states so that we can test them using our workloads.

7.4.2.2 CPU Load and Levels

Because the primary goals of this work are to show the promise of AC and the performance of Datacomp in the real world, I have chosen to monopolize CPU resources in a predictable and easily understood way rather than attempting to design "realistic" CPU workloads (which is its own contentious research area). Specifically, I use a contention script that reads pseudorandom data from `/dev/urandom`, compresses it using bzip2 (a fairly costly compressor) and discards the output by writing it to `/dev/null`. The script can generate any number of parallel threads. In this way, we can generate an arbitrary number of processes that essentially monopolize a single computational unit (i.e., core), reducing the amount of computational cycles available for compression and decompression in a consistent and incremental fashion. For experimentation, I primarily run tests with no contention and with two threads of contention.

7.4.2.3 CPU Frequency and Levels

My experiments treat frequency in one of two ways – manually controlled or dynamically controlled by the kernel. In the LEAP experiments, the CPU frequency is explicitly set and maintained in one of two states using command line utilities. The LEAP system used only supports two frequencies, 1.6Ghz and 2.4Ghz, and these can be set manually in order to observe the effect of frequency on Method performance.

For most of the other experiments, frequency is not controlled experimentally, but instead by the kernel's "on demand" mechanism, which attempts to provide the best performance while also minimizing energy and heat. While the frequency is not manually controlled in these tests, it is monitored for use in experiments.

7.5 Data Sampling and Scenarios

Scenarios for testing are defined by choosing a type of data, sampling the data in some way and setting the necessary environmental parameters. Experiments are then performed on those scenarios. These experiments can attempt to empirically determine the best greedy strategy (i.e., when using Comptool) or to test the performance of an AC system (i.e., Datacomp).

7.5.1 Sampling Data Sets

Each data class contains hundreds of megabytes of data. It is impractical to perform tests on all the data, so I required a method for sampling the data that preserved the important properties of the data as much as possible, while keeping the size small. I also wanted to sample the data in such a way that an individual "sample set" would have the same properties as a random "glimpse" into a stream of a particular kind of data as might be seen in transit between two communicating parties. This turned out to be more complicated than expected, because each choice had unintended consequences that affected compression outcomes. These are important issues for compression research (both adaptive and non-adaptive), so I include a discussion here.

7.5.1.1 Truncating Files and Random Sampling

My first method took a collection of files and truncated every file longer than a megabyte. Then, to create a sample set, I randomly picked files until the total size of the selected files was greater than one megabyte. This is an extremely simple and transparent mechanism. Unfortunately, there is a subtle side-effect of this approach that changes the compressibility properties of large files. The "truncation approach" keeps up to the first one megabyte of any larger file, using it to represent the entire file, regardless of the length or properties of the original file.

Unfortunately, the first portions of many files are simply not representative of the remainder, perhaps especially for large, difficult to compress data such as images or audio. Many file types, such as JPEG images with JFIF or EXIF data or MP3s with similar metadata, have "plain text" headers at the beginning of a file. In simple spot-checks of three MP3s and two JPEGs, I found a 1K header in a JPEG, and an 8K header in an MP3. The other files had headers smaller than 1K. While the effect may be small, it was still noticeable, in part because the quantization used by Comptool looks at input chunks as small as 32K. An 8K header would be a full 25% of that piece of input.

In an attempt to avoid this issue, my next approach was to sample data randomly from source files rather than to truncate them. In this method, I randomly sampled up to one megabyte from individual files in the pool, collecting data until at least a total of one megabyte was sampled. This approach avoided the “header bias,”, but still had significant drawbacks.

Both of these approaches result in sample sets with a wide size variance because rather than collecting only one megabyte of data, they collect *at least* one megabyte of data. Consider a run that first picks a 999K file followed by a 1M file – for a total size of 1.999M.[75] Another run might pick one one-megabyte file resulting in exactly one megabyte. This makes comparing times/results between sample sets difficult.

Finally, both of these methods picked files randomly from a list of names. This treated every file equally – which seemed like the right behavior at first glance. However, this is inaccurate if you're trying to get a representative “one megabyte sample” from the files. If you pick files randomly from a list of names, each file is equally likely to be selected, regardless of its size. Instead, if the sample is to represent a random subset of the bytes in a realistic may, files should be weighted by their size. The larger a file is, the more likely it should be that it appears in the sample.

7.5.1.2 Weighted Random Samples

The current design for sample selection is as follows. I pick files from a source pool at random and with replacement but weighted by size. For the first file only, I start at a random offset into the file, which simulates picking up "in the middle" of a hypothetical file transfer. Then, I read up to some *N* bytes (currently one megabyte). If I fail to acquire *N* bytes, I keep the partial result of size *S* and try again with a new file randomly selected in the same way. This file is read from the beginning, up to *(N – S)* bytes. If it so happens that this file is less than *(N-S)*

[75] The truncation approach would still result in 1.999M, since the first file would not be truncated.

bytes (i.e., we have not yet read a total of *N* bytes), we read additional files up to the remaining number of bytes.

In the end, this creates a sample set with the following properties:

1. Files are not arbitrarily or unrealistically truncated
2. Files are weighted by size so that they appear in realistic proportions
3. The resulting sample sets will all be exactly *N* bytes (e.g., one megabyte)
4. The files remain discrete (they are not concatenated together)
5. Each unique sample set is drawn from a random permutation of the files

This is the mechanism used for sampling the data in this work.

7.5.1.3 Persistent Sample Sets

In early testing, I generated temporary file sets for each test. As a result, one test run for a particular scenario X_1 would have a set of samples S_1 while a subsequent run in a slightly different scenario X_2 would have a completely different set of samples S_2, and so on. This is experimentally acceptable, but it means that outcomes from one test to the next are not directly comparable – the data is only useful in the aggregate. It also means that Comptool would need to be run on every sample set used in every test in order to find and compare the best potential strategies at the Method level.

For this work, I split each data source into two portions, one for training and one for testing. I then created 50 numbered 1M sample sets for each data source. The same subset of these sets are used as the data for each test in every environment, which facilitates the comparison of results for the same data type but performed in different environments. The training data is used for testing and development of Datacomp's Models, while the test data is used only for actual adaptive or non-adaptive tests.

7.5.2 Scenarios

Scenarios define the long-term task at hand and are a combination of some data or type or data and some environment.

For example, one scenario would be:

- 25 one megabyte samples of Wikipedia data
- 100mbit/s network
- 50% CPU load
- Dynamic CPU frequency

Tests take place "inside" Environments and tell us something about how compression can affect efficiency in that Scenario. For example, using Comptool in this Environment would identify the average best greedy strategy for the Scenario through the use of exhaustive search. Executing drcp in this Environment would task Datacomp with attempting to optimize performance in this Environment in real time. Testing dzip in this scenario would use a greedy brute force search (like Comptool) to find the Strategy with the best compression ratio.

For these tests, a set of samples was created ahead of time (as described in Section 7.5.1.3). Since the Data and Environments would be the same in both tests, we can compare the results. For example, to evaluate drcp's performance versus a static strategy, we could compare the average adaptive results to the average cost of drcp with a fixed strategy (e.g., always using zlib at strength 1 with two threads). To compare drcp's performance to a greedy best solution, we could compare drcp's average adaptive results to the average "best greedy results" obtained by Comptool.

7.6 Discussion

Creating data workloads for designing and evaluating Adaptive Compression systems is a challenging task with potentially competing criteria. This section has discussed the sources and sampling of data as well as the construction and use of environments and scenarios. Variations in these concepts are described in more detail in the results section of each tested component (Sections 8, 9 and 10).

While these workloads might actually be the most diverse set of test material used for AC experimentation and evaluation to date (at least in the context of "real world" data), no set of workloads can be perfect because there are simply too many potentially relevant variables that might be significant. Performing additional experiments on existing data with different environments and bandwidths is easy enough; it usually only requires configuration changes. However, performing tests with different data types requires identifying, collecting and sampling this data in a defensible and unbiased manner. This is challenging and time consuming even for one type of data, and it is difficult to say how many meaningfully unique types of data exist from the perspective of compressibility and compression performance.

Ultimately, any set of tests and test parameters is only as good as it is representative of the target environment. However, identifying the "typical workload" of a class of target machines is a research project in itself. For this reason, I have primarily focused on treating the Data types and Environments as discrete tasks for typical machines, letting the data speak for itself, rather than presupposing that a specific "workload mix" is most common. In other words, I generally report results such as "Facebook data at 1Mbit/s with 4 CPU contenders" or "YouTube data at 100Mbit/s at CPU frequency X" rather than creating a single workload consisting of some proportions of each discrete task or assuming that every data type and environment is equally likely. This is especially important for compression experimentation because the data (and bandwidth) in question is so critical – for example, the potential value of compression for “web traffic” depends significantly on what proportion of the traffic is Wikipedia, Facebook and YouTube data (etc.), and how quickly that data is expected to arrive at the receiver.

While evaluating these elements separately does not result in a convenient single performance value (“My system saves X%!”), it provides direct access to the statistics that are most interesting to other researchers, who can then look at the individual scenarios and determine how well (or in what proportion) the experiments match the systems they are interested in. It also future-proofs the experiments somewhat in that if a new type of workload becomes important, it can add to the experimental results but need not throw their validity into question. At the same time, results can easily be aggregated if that is important for a particular purpose.

Finally, in any experimental program centered on compression, researchers must take great care in collecting, sampling and managing workload data. In terms of collecting, some human verification is of the resulting data necessary to ensure that the data actually is what is intended. For example, researchers should verify that it is uncompressed, not encrypted, has networking headers removed (or not, depending on your test), and so on. Assuming that your collection mechanism is right can unwittingly lead to headaches – or worse – bad results.

Similarly, the choices made regarding sampling and input sizes are critical for making experiments practical and for testing various environments. However, these choices can (like collection procedures) easily and inadvertently have significant impact on test results which may be difficult to detect. For these reasons I highly recommend keeping all original source material so that sampling procedures and experiments can be re-run as needed.

Ultimately, I believe that well-rounded, realistic and defensible workloads are critical for applied compression research. If workloads are too easy, the results will be unrealistically good. If they are too hard, the results will be overly pessimistic. Too exotic and they will not be relevant. AC researchers will make a more significant and longer lasting impact with their research if their experimental workloads are compelling and carefully designed.

8 DZIP RESULTS

8.1 Overview

Dzip is fundamentally a utility wrapper around Datacomp, intended for use on file data. Since dzip is based on Datacomp, it can perform manually requested compression operations and optimize for time/energy or space. However, since file compression is typically used to save space, the primary use for dzip is as a demonstration of Datacomp's "space" decision Model.

This Model performs the same "greedy best" search used by Comptool, with virtually the same set of compression Methods. However, while Comptool performs all operations and discovers the best strategy after the fact, dzip uses the greedy approach in an online fashion. To do this, dzip performs each compression Method at a given offset into the file. It writes the compressed data from the result with the best compression ratio to the output and advances the cursor into the input by the number of (uncompressed) bytes consumed by the best Method. It then repeats the process until the file is completely compressed. (See Section 5.4.2.2 for a more complete description of the "greedy best" strategy.)

Because Datacomp and Comptool use the same greedy algorithm and the same compression Methods, we can compare the results from Datacomp and Comptool to ensure that Datacomp's online "space" Model is working properly. This will also highlight the overhead of the Datacomp Frame Headers (DCFH), that is, the data in the protocol that specifies how a given input was compressed.[76] In this section, we discuss dzip's performance goals and the Comptool experiments used to evaluate dzip's results.[77]

[76] Since Comptool simply performs compression operations, it does not pay this overhead.
[77] Data and other information regarding these experiments will be available at http://labs.tastytronic.net/datacomp/

8.2 Goals

The high-level goal of dzip is to get as close to the maximum compression for a given input as possible, without having to manually select the best compression Method. I also thought it was plausible that a greedy, quantized approach as used in dzip might actually outperform static strategies (using one Method on the entire file) since dzip would be able to optimize compression specifically for various subsets of the input. I think of the first goal – being close to the best without needing manual intervention – as the “core” goal of Datacomp, while the second goal, actually outperforming static compression as the “bonus” goal, since it may not be possible.

It was not clear ahead of time whether either of these goals were possible. I thought it was possible that a quantized approach could beat static approaches, but I also thought it possible that a quantized approach would hurt compression ratios too severely. This uncertainty partially inspired the design and evaluation of dzip, which is focused on showing two things: how well dzip manages to compress, and how much time achieving that compression takes. I designed the evaluation this way because even if dzip succeeds academically (i.e., achieves its core goal), the time cost would need to be manageable if dzip could be useful in the real world.

For the “bonus” goal to be met, dzip would need to *consistently outperform the average compression ratio of the best single strategy for a class of data*. For example, if the best average “static” CR for Wikipedia data is 0.25, dzip would need to achieve an average CR of less than 0.25, with non-overlapping confidence intervals. For the core goal to be met, we would be satisfied if dzip simply performed similarly to the best single strategy. (At that point whether the time cost was worthwhile is somewhat subjective.)

Because we do not know the distribution of data in the real world, we run separate tests on each of our various classes of data. At the very least, we would expect dzip to perform as well as the best single static strategy for the given input (the core goal); if gzip is the best static compressor, then dzip should at least do as well as gzip. Optimistically, we hope that dzip can consistently outperform the compression achieved by static strategies for *all* classes of data, rather than just some classes of data (the bonus goal). If dzip consistently meets its core goal, then it will at least arguably be a valid compression mechanism. If it achieves its bonus goal then it would be an obvious choice for truly compression-critical tasks.

On the other hand, if dzip only achieves its core or bonus goals for *certain* classes of data, that is, if it consistently performs (even slightly) worse for certain inputs, then dzip would be much less generally useful. Perhaps, if performance on a particular data class was significantly better than any other static strategy, dzip might have use as a niche or data-specific compressor, such as NCTCSys [59]. But in practice, niche compressors are only known and used in extremely specific environments, and it seems unlikely that such a complicated mechanism would be worthwhile for such an application.

To summarize, the goal of dzip is to be a high-quality, general purpose compressor that meets or exceeds the compression ratios achieved by the best static strategies for a broad range of data. If the mosaic-like "greedy best" strategy used by dzip to optimize compression ratio is successful, it could be broadly useful in the real world and might inform compressor design in the future. On the other hand, if it is not successful, or if it is successful but is too costly, it is still an interesting result but is unlikely to find use in the real-world due to the high computational cost of its brute-force mechanism.

8.3 Experiment Design

At a high level, evaluating dzip requires performing adaptive compression dynamically on sets of sample data from each data class using dzip, and comparing it against the results obtained by Comptool. In addition to Comptool, I also run the standard compression utilities on the file (e.g., gzip, bzip2, etc.), which allows me to compare dzip and Comptool's results against static strategies as they exist in the real world (dzip's "competition"). This is important, since the quantization used by both Datacomp and Comptool could have impacts on compression ratio results in particular. Additionally, using the standard utilities allows me to test strength levels of gzip and xz that are not used by either Comptool or Datacomp (e.g., every gzip and xz level between 1-9). While we focus on the compression ratios achieved by these Methods, it is also interesting to look at the resulting compression and decompression times as they vary across Methods and strength levels.

As described in Section 7, I created 50 one-megabyte samples drawn from each data class. Each of these "sample sets" is composed of one or more files. However, when compressing for space, having many files of different sizes makes comparisons between single tests or across data classes more complicated, because it may not be clear if a difference in performance was due to the data or Method used, or due to the fact that one test processed *n* files of various sizes while another test processed *m* files. Because of this issue, I concatenated each sample set together without gaps, using the utility "cat". This means that, unless otherwise specified, each sample set used for dzip evaluation (for every data class) consists of exactly one megabyte of data drawn from each class, and tests were performed over 50 sample sets (one test per sample set). Since the distribution of compression ratio means was not known for each combination of data and Method (and have been observed by me to not always be normally distributed), I computed the confidence intervals using the BCA bootstrap method at the 95% confidence level.

The bulk of dzip tests were performed using these 1M sample sets as just described. However, since compression ratio can often significantly improve with greater amounts of input, I performed some tests where I concatenated all 50 sample sets from each class into one large 50-megabyte text. These tests are described in Section 8.8.

Concatenating sample sets into one-megabyte (for the typical tests) and 50-megabyte (for the special tests) chunks reduces the representivity of the test data in some sense, because in doing so data from completely separate files are combined and treated as a single input. However, due to the importance of input size for compression ratio, and since in other tests I use the files separately as they appear in the sample sets, I felt that concatenating the files together was a justifiable choice. That said, researchers must be thoughtful when they combine test data for compression workloads, lest they unexpectedly change the compression properties of their test data. I discuss these issues in the remainder of this section.

When concatenating data for compression testing, small differences in approach can result in unexpected effects that change the compressibility of the data. For example, the "tar" file format is sometimes thought of as a simple concatenation of all files. While it is in fact a concatenation of all files, it is not *only* a concatenation of all files. The tar file format includes metadata for each included file, such as file names and permissions. Additionally, there is (according to the standard) a footer at the end of a tar file, which can be quite large (about 10K on my machine). Normally, tar files are compressed (e.g. with gzip or bzip2) and these headers are highly compressible. As a result, this overhead is not a concern in most applications. However, if tar is used to concatenate files for a compression study, it will actually make the data more compressible *in proportion* to the original data.

Filename	Size
random.10K.1	**10,240**
random.10K.1.gz	10,276
random.10K.1.tar	20,480
random.10K.1.tar.gz	10,468
random.10K.2	**10,240**
random.10K.2.gz	10,276
random.10K.1-2.cat	20,480
random.10K.1-2.cat.gz	20,518
random.10K.1-2.tar	30,720
random.10K.1-2.tar.gz	20,740

Table 16. The utility 'tar' expands input data due to its file format.

This effect is shown in Table 16 and described here. The files `random.10K.1` and 2 (in bold) are 10K (10,240 bytes) of random data taken from `/dev/urandom`; they should not be compressible. Gzipping either file results in 10,276 bytes (e.g., `random.10K.1.gz`), for an overhead of 36 bytes. A tar file created with `random.10K.1` as its input (i.e., `random.10K.1.tar`) results in a tar file which is 20,480 bytes – double the size of the 10K input. Compressing that tar file results in 10,468 bytes, because the metadata compresses relatively well. Still, this is 228 bytes larger than the raw input `random.10K.1`, and 192 bytes larger than the gzipped original file (`random.10K.1.gz`) since there is now extra data in the file.

Concatenating both files using "cat" results in a single file (`random.10K.1-2.cat`) which is 20,480 bytes – exactly the sum of the parts. Gzipping this file results in 20,518 bytes (`random.10K.1-2.gz`), a total overhead of 36 bytes. In contrast, using tar to concatenate the two files results in a "tarball" of 30,720 bytes (`random.10K.1-2.tar`), or 10,240 bytes greater than the sum of its parts. Compressing this tarball results in 20,740 bytes (`random.10K.1-2.tar.gz`), which is a compression ratio of 0.32 *over the tarred input* (`random.10K.1-2.tar`) but at the same time is also 260 bytes larger than the concatenated input `random10K.1-2.cat`, and 222 bytes larger than the `random.10K.1-2.cat.gz`.

Accordingly, the solution is to use cat or a similar utility to append the files together without breaks. While this is obvious in retrospect, the potentially significant effect of simply using tar rather than cat makes this an important cautionary tale.

8.4 Compression Ratio

The average compression ratios of dzip and a variety of standard compressors for each data type are shown in Figure 17 through Figure 25. These graphs depict how well dzip does in comparison to *standard compressors*. Standard utilities were used (rather than comparing against dzip using a static strategy) since they are the current real-world alternative and because it was plausible that standard compressors would benefit from compressing the entire input at once, rather than quantizing it as Datacomp does. Since I used the standard compressors as comparisons, I also included all gzip and xz levels to show the relative effect of these strength levels. (Datacomp only uses gzip and xz levels -1, -6, and -9.). Overplotted numbers display the average CR and confidence intervals show the 95% confidence level taken using the bootstrap method over results for all 50 one megabyte samples compressions for each Method.

As detailed in Section 8.2, we would ideally like to see dzip meaningfully out-compressing the alternatives for each data type, although this is the best case scenario. It would be acceptable if dzip merely does as well as the best compressor and it is not the case that a single algorithm is always the best compressor. If this were true, we could then say that dzip is capable of always getting the “best CR” without inspecting the data. A disappointing result would be if dzip performed significantly worse than the best compressor, since this would disprove our hypotheses regarding the ability to maximize compression by compressing discrete chunks. In the remainder of this section, I analyze dzip's performance on each of the nine data types.

8.4.1 Binary

Binary data (as shown in Figure 17) serves as a good introduction to these results as it is similar in many respects to other CR charts in this sequence. Perhaps the first thing we notice is that LZO is statistically worse than the other Methods in terms of compression ratio, with an average CR of 0.580 versus the best CR of 0.376 shared by xz -6 through -9. The confidence intervals (CIs) for all other Methods overlap, with the exception of gzip -1, where the 0.108 difference in CR between gzip -1 and xz -9 is also statistically significant.

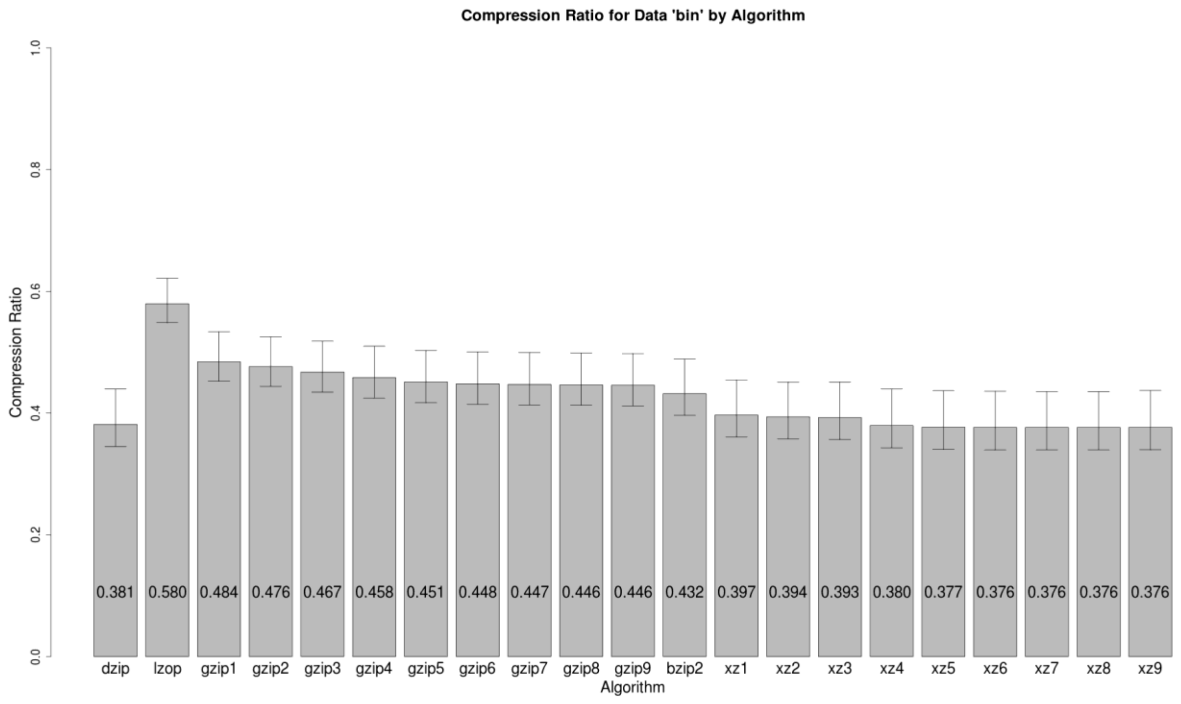


Figure 17. Average CR for dzip and static compression Methods on the "binary" data set (1M samples).

Dzip achieves a CR of 0.381, which, due to the overlapping confidence intervals (CIs) and means, is not a statistically significant difference from xz -9's CR of 0.376. Thus, we can say that dzip fulfills our core goal of doing about as well as the best choice, although it does not meet our bonus goal of out-compressing the best standard compression utilities.

8.4.2 Mail

Figure 18 shows the average CR for mail across our set of Methods. LZO is the clear outlier (average CR: 0.967), unable to achieve much compression whatsoever on this data. Dzip's average CR actually beats xz -5 by two tenths of a percent (for a CR of 0.713), although the CIs overlap for these and all other compressors for this data set. Nevertheless, we can say that dzip achieves the best average CR for this data. We also see that increasing the upper strength levels of gzip (except for gzip -9) and xz fail to improve the CR meaningfully, and that the lower levels of xz do worse than all other Methods, except LZO. While these differences are not generally not statistically significant (the small xz differences *may* be), the fact that patterns visible across the various related Methods suggests that there are genuine effects taking place, and illustrates that simple assumptions about the effect of strength levels are naïve.

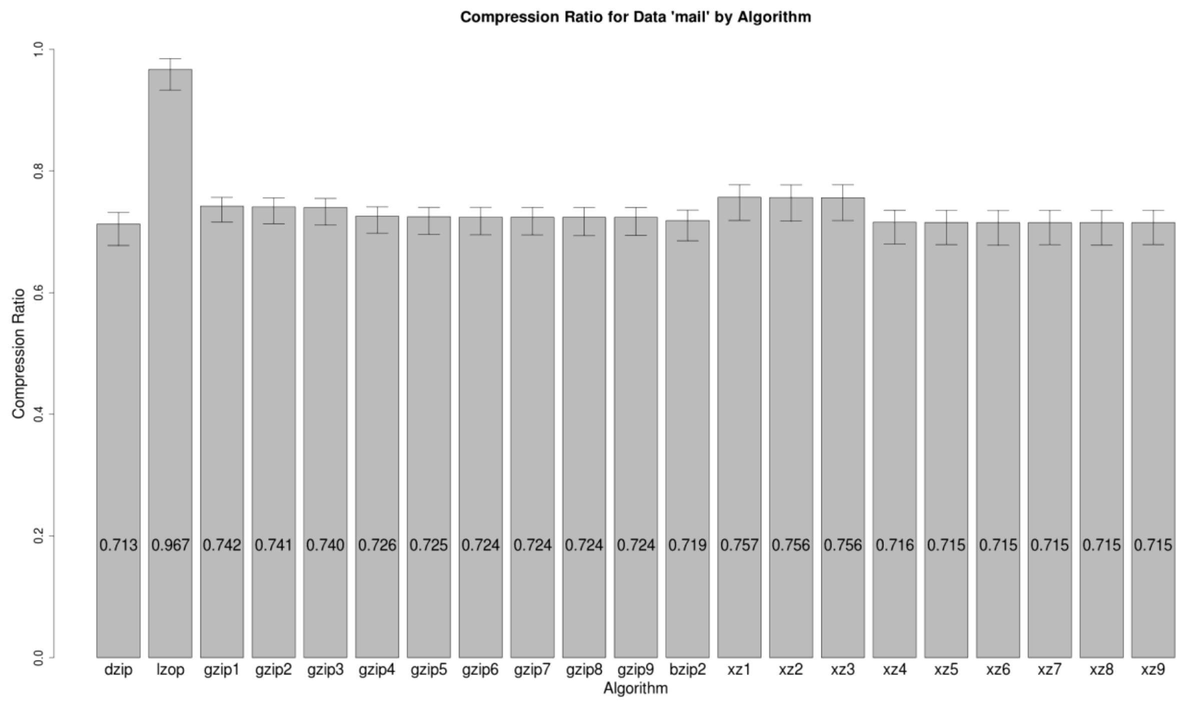


Figure 18. Average CR for dzip and static compression Methods on the "mail" data set (1M samples).

8.4.3 Wikipedia

Wikipedia data continues these general trends as shown in Figure 19, with some minor variations. All CIs overlap, except the mean CR for LZO (0.457) is statistically worse than several compressors. Specifically, dzip (0.311), bzip2 (0.318), and the xz Methods (0.318-0.301). Xz -6 through -9 share the best CR, at 0.301. In contrast to mail data shown in Figure 18, where xz -1 through -3 were outliers, all xz Methods outperform all non-dzip Methods for Wikipedia data, suggesting that differences in the characteristics of the data do affect CR outcomes in unpredictable ways. While dzip is not the best compressor for this data, it is the closest compressor to xz's result (and the differences are statistically insignificant). As a result, we can say that dzip meets the core goal.

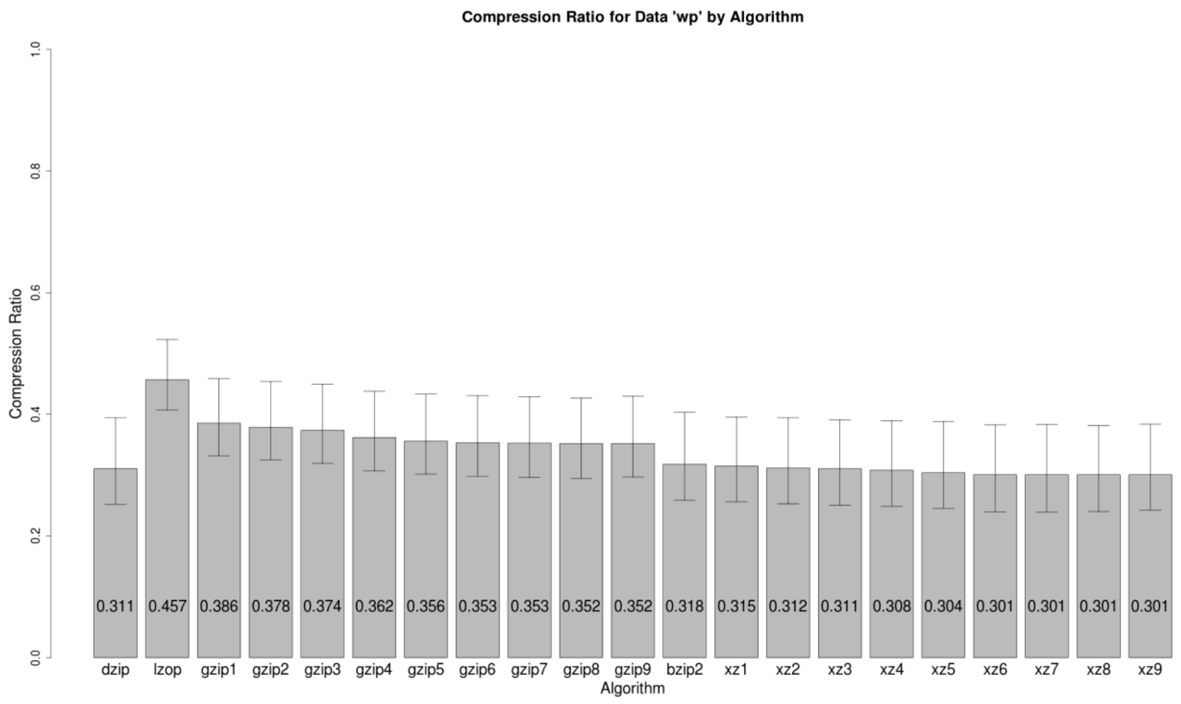


Figure 19. Average CR for dzip and static compression Methods on the "Wikipedia" data set (1M samples).

8.4.4 Facebook

The CR data for Facebook (shown in Figure 20) is similar in “shape” to the Wikipedia data, although with worse CRs and proportionally closer averages and CIs. Accordingly, all confidence intervals overlap. Again, dzip (0.622) does not have the best average CR (xz -6, 0.617), but it is the closest to the best that is not another xz compressor. Because of this, we can say that dzip meets its core goal for Facebook data.

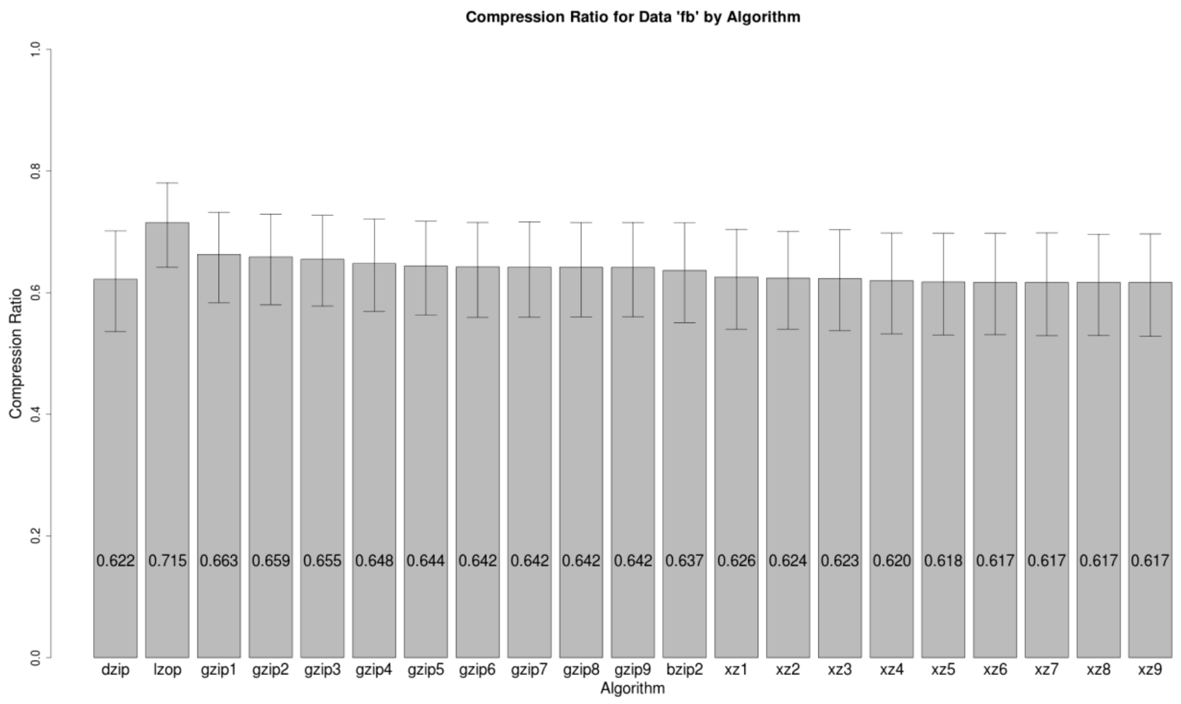


Figure 20. Average CR for dzip and static compression Methods on the "Facebook" data set (1M samples).

8.4.5 YouTube

CR data for YouTube (Figure 21) illustrates the performance of compressors on data that is only very slightly compressible. Here, all confidence intervals again overlap. dzip shares the best compression ratio (0.970) with gzip -4 through -9. Xz is slightly worse, at 0.972 for levels -4 through -9. As a result, we can say that dzip again attains our core goal.

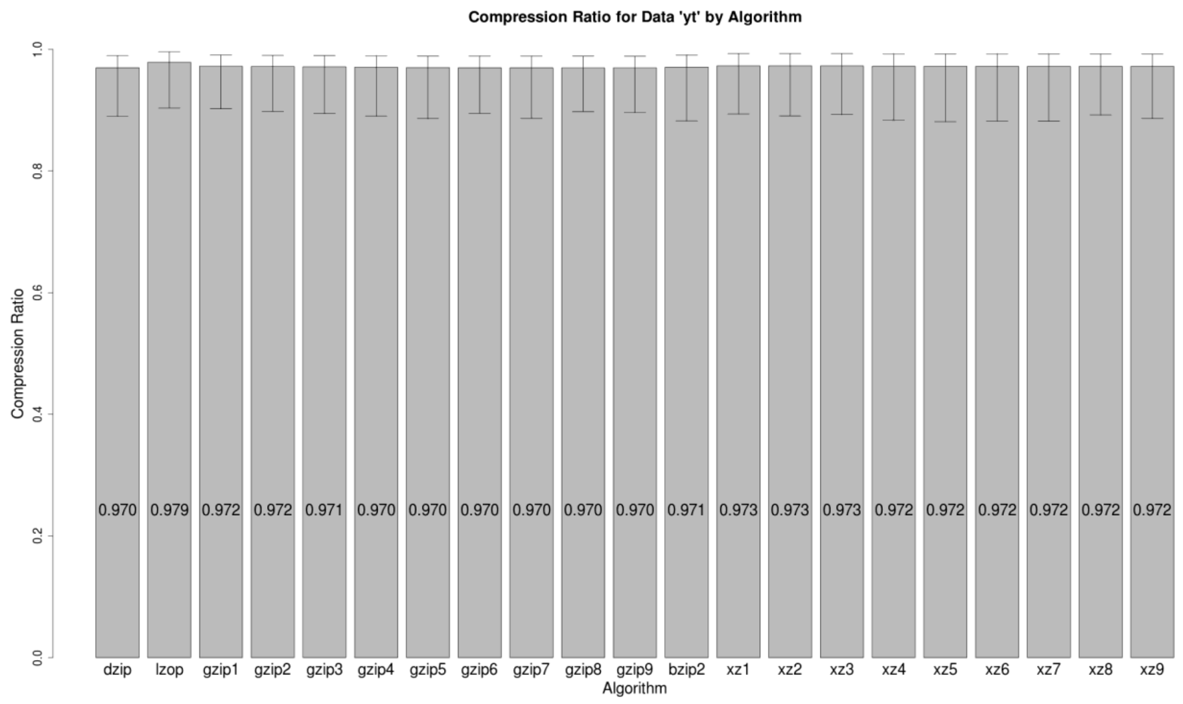


Figure 21. Average CR for dzip and static compression Methods on the "YouTube" data set (1M samples).

8.4.6 User Web

The results for User Web data, shown in Figure 22 are very similar in shape and value to the Facebook data. Xz is again the best compressor (xz -6 in this case) with a CR of 0.659 as opposed to dzip with 0.673. Once again, all confidence intervals overlap, and dzip beats the other compression algorithm families (i.e., LZO, gzip and bzip2).

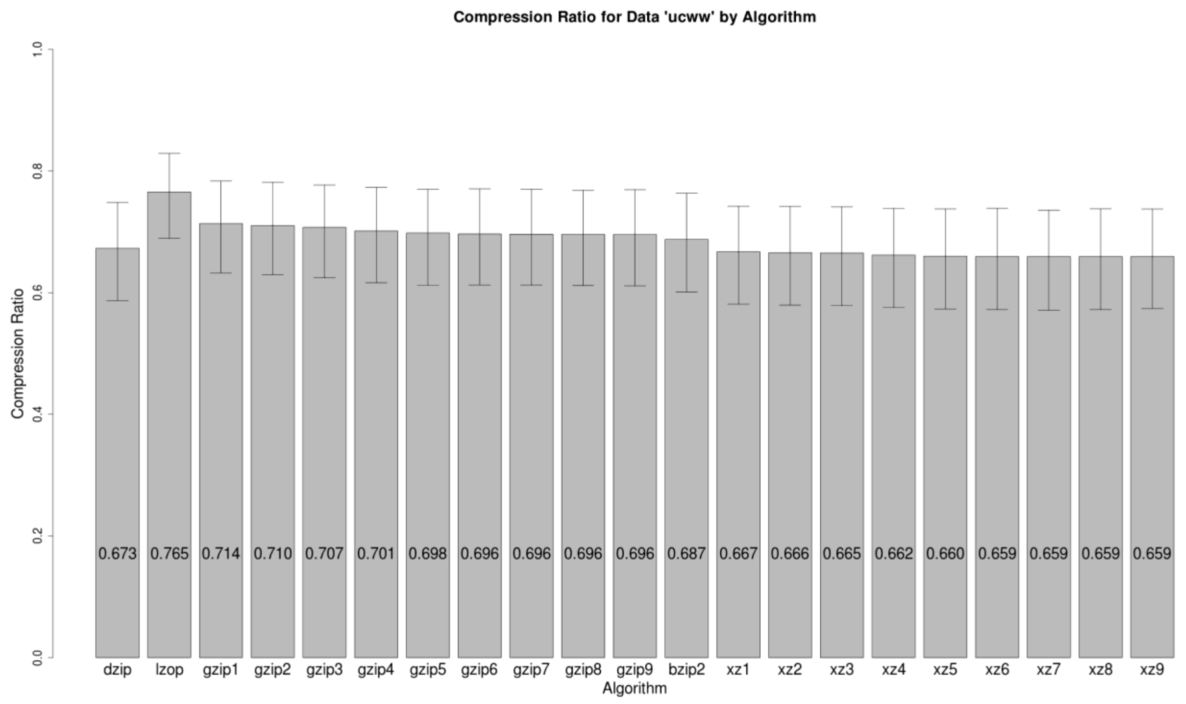


Figure 22. Average CR for dzip and static compression Methods on the "User Web" data set (1M samples).

8.4.7 User File

User File data is most similar to YouTube data in its achieved compression ratios. This is likely due to the fact that much user data (i.e., images and music) is already compressed using a lossy Method similar to streaming video. As a result, no compressor does especially well. However, CRs in general are better for User Files than YouTube, probably because of the additional portion of input data that is compressible. All confidence intervals overlap, and dzip (0.938) is the second best Method family next to xz, where xz -4 achieved the best CR (0.928).

Finally, when discussing User Files, it is important to note that this data is completely unfiltered and reflects the large proportion (in bytes) of multimedia data stored on user machines, even though in practice many of these files (e.g., vacation photos from five years ago) are rarely accessed. It would be interesting to perform tests on user data based on frequency of access, although that is left for future work.

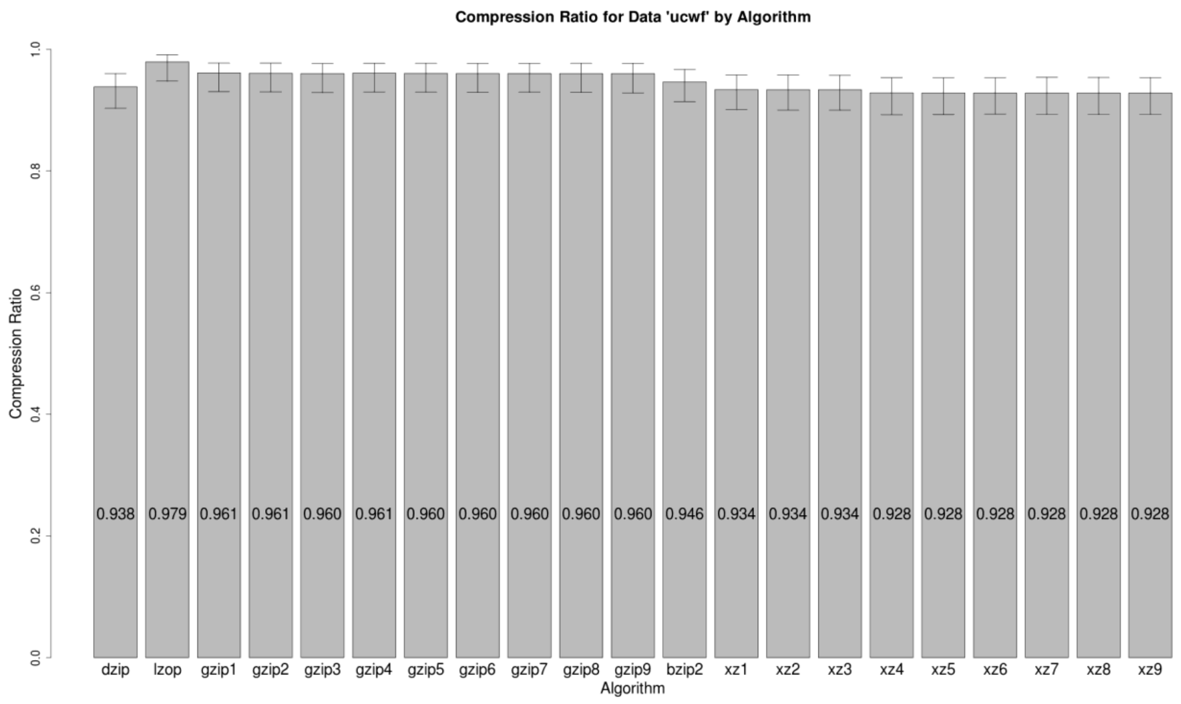


Figure 23. Average CR for dzip and static compression Methods on the "User Files" data set (1M samples).

8.4.8 Zero and Pseudo-random

Last but not least, I do not show the compression ratios achieved on zero and urandom data in graph form, because they are not especially interesting. All compressors do incredibly well on zero data, and all compressors fail on urandom data. Still, it is worthwhile to summarize the results for posterity.

Bzip2 (0.00004) is the "best" for zero data. Dzip (0.001) is *not* the second best compressor because of the header overhead required for Datacomp Stream Frames. Instead, the second best compressor is xz -1 (0.0002). As a result, dzip does not technically meet either of our goals for zero data, although given the incredible performance on this data (and its artificial nature), the fact that dzip is not the best or second best seems relatively insignificant. Dzip (1.00006) *is* the best for urandom data, because it detects that the data is uncompressible. These differences are statistically significant for reasons which are described in the analysis for the next experiment.

8.5 Validation with Comptool

As shown in Figure 24, except for zero and urandom data, the average compression ratios achieved dynamically by dzip are statistically indistinguishable from those composed by analyzing Comptool data. For zero and urandom data, there are very slight differences in CR means and no CIs, which will be explained later in this section. As a result, we can say with confidence that dzip indeed performs the greedy algorithm properly.

That said, while the differences are not statistically significant, there are differences. This seems like it should not be possible, since dzip and Comptool use the same underlying algorithms. Investigating them illuminates some of the inner workings of both systems.

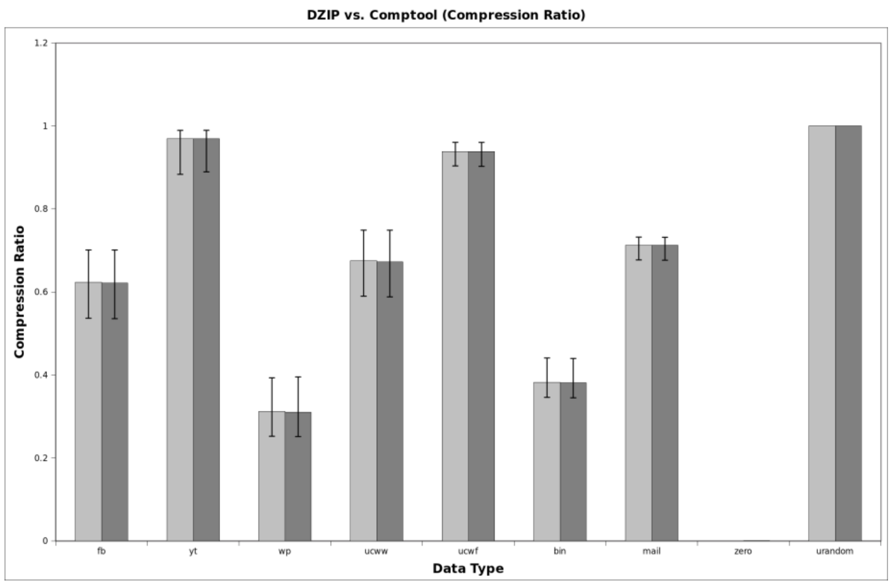


Figure 24. Comparison of compression ratio achieved by dzip vs. Comptool.

Inspection of the underlying choices and results for both Comptool and dzip reveals two sources of "error" that can lead to small variations (about 1/10th of one percent of CR or less) between the two even though they use the same compressors and greedy algorithm. The first source is that while the underlying algorithms used by both tools are the same, there are slight differences in the output formats of the two systems. The second source of error is due to differences in rounding used by the two systems.

In terms of output differences, dzip must include a dzip header on the file, in addition to a Datacomp Stream Frame Header (DCSH) for each Datacomp Frame (DCF) in the output stream. These headers, even though they are small, can affect the compression ratio of the final result slightly. On the other hand, Comptool only performs the compression for each chunk, without any headers or other metadata.

The rounding differences between dzip and Comptool are mundane, but they can appear in results. Comptool's data is generated by Python and stored in an SQL database, so compression ratios are written as floating-point numbers with high precision. It is unlikely that two Methods will tie, unless they are actually using identical mechanisms. Alternatively, Datacomp calculates compression ratio only to three digits of precision. For Datacomp, a CR of 1,000 is the Comptool equivalent of the CR 1.0. And a Datacomp CR of 1 is a Comptool CR of 0.001. This design choice was made for Datacomp for two reasons. First, I wanted to avoid floating-point numbers as much as possible for speed. Second, if two chunks are equivalent in compression ratio out to three digits, I thought it would be worthwhile to "break the tie" using our other mechanisms (i.e., chunk size and output rate).

When these two issues are combined, it can lead to the very slight variations in compression ratio as seen in this data. This observation also leads us back to the two classes of data that have significant differences – zero and urandom data. The differences in average CR for these two classes are statistically significant only because the outputs are of slightly different size and there is *no variation* in the strategy for either system. Comptool is able to simply use "null compression" for urandom data which pays no overhead at all in terms of headers or metadata, while dzip must annotate the Datacomp Stream Frames, even though dzip also uses null compression. As a result, where Comptool achieves a CR of 1.00000, dzip's best CR for urandom is 1.00006. Similarly, Comptool's CR for zero data is 0.00005 (90 bytes) while dzip's is 0.001 (1,126 bytes). While these differences have statistical significance, they are not likely to have practical significance in the real world.

8.6 Time Cost

Dzip is extremely slow to compress, because it represents the sum of every tested compression operation performed by dzip. For these experiments, that includes the compressors null, lzop, gzip (1, 6 and 9), bzip2, and xz (1, 6 and 9) and the chunk sizes 32K, 64K, 128K, 256K and 512K (at every applicable 32K offset as described in Section 5.4.1). Dzip doesn't need to do as many compression operations as Comptool, however, since certain choices made by dzip preclude possibilities that Comptool checks because of its brute force design.

For example, Comptool compresses every 512K span in the input starting at 32K intervals. In contrast, dzip will compress a 512K chunk at offset 0 (in addition to all other chunks starting at 0). However, it won't compress a 512K chunk starting at offset 32K unless it turns out that the 32K chunk starting at offset 0 is the best – at which point dzip will start new compressions at offset 32K. If the best chunk turned out to be 256K, the next set of compression tests would start at 256K, skipping many potential compression operations. Thus, while dzip's time cost is extraordinary, it is "only" tens of times worse than a static compressor, rather than hundreds or thousands of times worse.

Compression times for dzip versus the best alternative are shown in Figure 25, while decompression times (versus the same alternatives) are shown in Figure 26. The alternative shown was chosen by looking up the timing for the static compressor with the lowest average CR for each data type. Almost always, the best static compressor was xz -9. Gzip -9 was the best compressor for YouTube and bzip2 was the best compressor for zero data. Dzip was actually the best compressor for email and urandom data, so to provide an alternative I chose the second best average for these types. The second best compressor for email was xz -9, while the second best compressor for urandom data was (surprisingly) LZO.

Not counting the extrema data (zero and urandom), dzip's compression time ranges from about 5 seconds to about 10 seconds per megabyte of data. As a simple example, the largest average compression time for a single data type and Method is for binaries with xz -9, which takes around 0.5s to compress each 1M sample set. For comparison, dzip takes nearly 10 seconds to compress a 1M sample. This is a time penalty of almost twenty times.

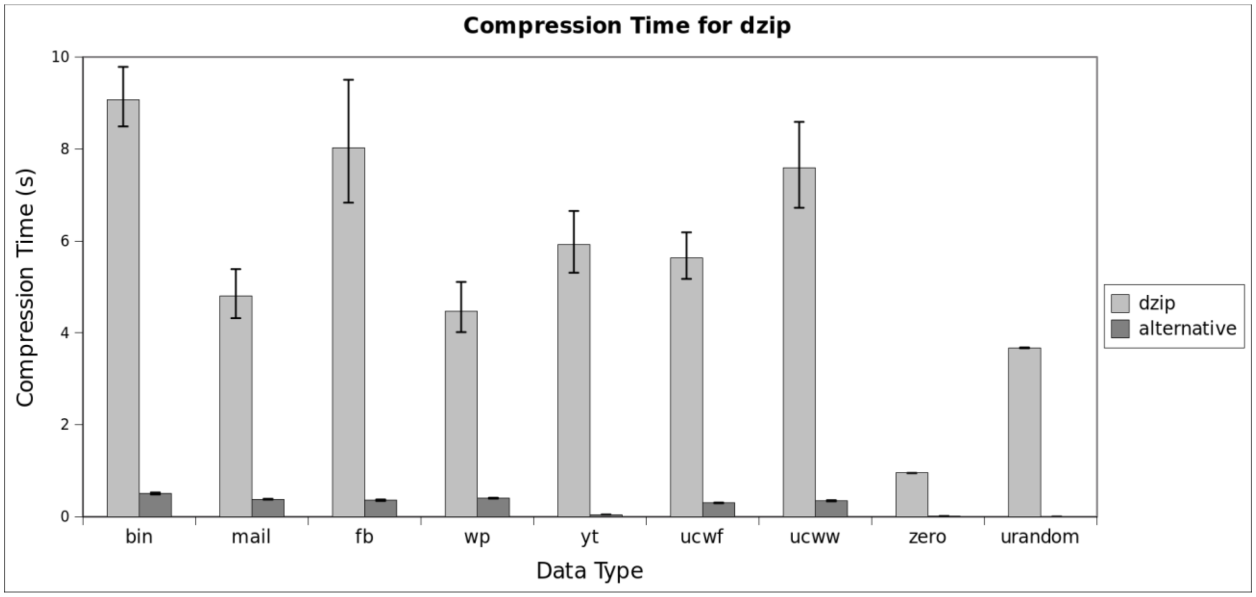


Figure 25. Compression time for dzip vs. the next best alternative.

Dzip's decompression times (as shown in Figure 25) are relatively fast in comparison because the Methods necessary for decompression are encoded into the Datacomp Stream – dzip does not need to "discover" them through search. These costs are on the order of individual compressors – typically they are slower than static utilities by some single-digit factor (as opposed to tens for compression). Decompression times for dzip are very consistent, with narrow confidence intervals.

Nevertheless, dzip's decompression is still slower than standard utilities. Static compressors (as tested here) simply apply one mechanism to the entire input so decompression is fast once started, while with dzip the stream is quantized and compressed with multiple Methods. When dzip decompresses a stream, Datacomp determines the decompressor required by reading the Datacomp Frame Header (DCFH) for the every frame in the stream – as rarely as every 512K bytes but potentially as often as every 32K bytes. This is not a slow process by any means, but it is a repeated overhead that static Methods do not pay, currently including repeated decompressor instantiation and memory allocation.

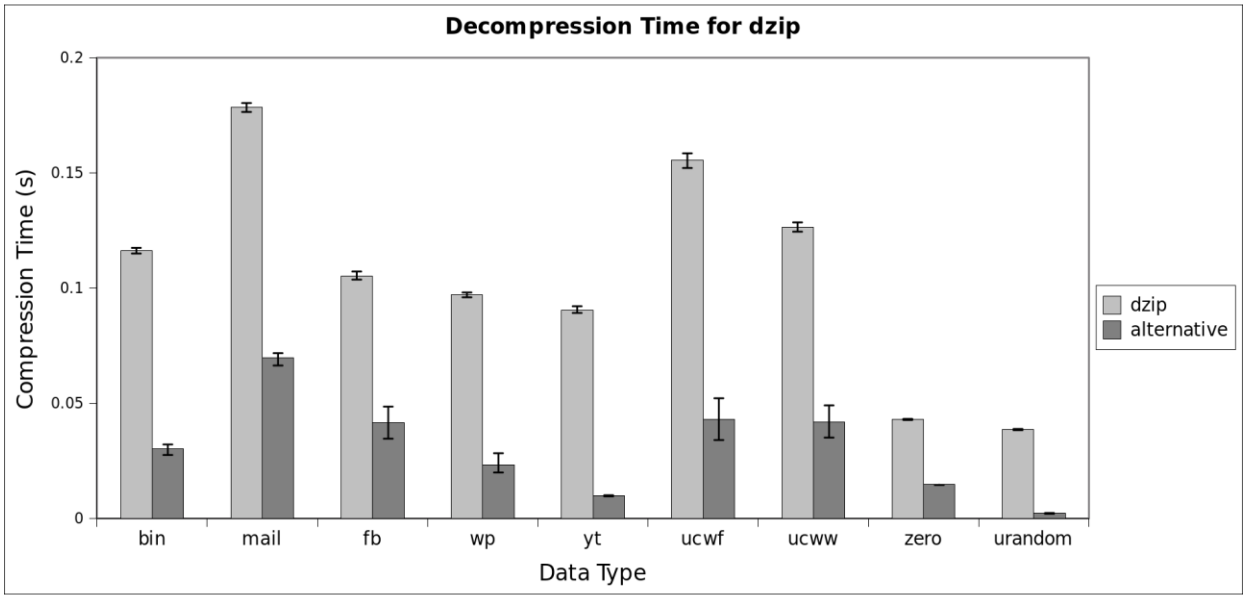


Figure 26. Decompression time for dzip by data type.

8.7 Performance vs. Goals

The more difficult goal for dzip was to compress more effectively than static compression choices through a fine-grained quantized approach. If this occurred in general in our data, we would see dzip being the "winning strategy" for a majority of the 1M sample sizes across all types. Unfortunately, this is not the case. Dzip is only unequivocally the best strategy for one type of file, pseudorandom data, included only as a worst case scenario. For other types of data, dzip is sometimes the best, or tied for the best, but the differences are not statistically significant. In many cases, some other static choice is better, even if dzip is comparable.

Instead of looking at CR means, a different approach is to examine which major compression Method was best for each of the 50 sample sets for each data type. Table 17 does this by tabulating the results of 50 runs of each data type. Each row is associated with a different data type, while each column is associated with a different compressor family (i.e., if xz -8 is the best for one sample set and xz -9 is the best for another, that counts as two "wins" for xz). The algorithm family with the greatest number of wins for a given data type is in bold. The total number of "wins" for each algorithm family is summed at the bottom, with all data types ("total") and with all types except zero and urandom ("no extrema").

Data	**dzip**	**lzo**	**gzip**	**bzip2**	**xz**
Binaries	*4*	0	0	0	**46**
Mail	*15*	0	1	**17**	**17**
Facebook	3	0	*7*	2	**38**
Wikipedia	0	0	0	*1*	**49**
YouTube	0	0	**45**	*3*	2
User Web	3	0	*8*	2	**37**
User Files	11	0	*12*	**16**	11
/dev/urandom*	**50**	0	0	0	0
/dev/zero*	0	0	0	**50**	0
TOTAL	*86*	*0*	*73*	*91*	***200***
**No Extrema*	*36*	*0*	*73*	*41*	***200***

Table 17. Count of which of 5 algorithms were best in terms of compression for 9 data types. Each row represents 50 distinct 1M samples compressed by all algorithms.

Xz is clearly the best compressor in terms of compression ratio, out-compressing or tying another compressor on five out of nine data types (rows). Bzip2 came in second as the best choice for three data types (including a tie with xz for email). Gzip and dzip both were best for one data type each, for YouTube and random data samples, respectively. As described previously, dzip performs well for random data because it is able to send it uncompressed with only a small header every 512K bytes.

Looking at the totals, xz wins *200 out of 450 times*, more than twice as often as bzip2 with 91 (the next closest algorithm). Dzip is third with 86 wins, followed by gzip with 73 wins. LZO never wins. However, while dzip is especially effective with urandom data, and this could represent any type of uncompressible data (e.g., previously compressed or encrypted), urandom and zero data are somewhat artificial scenarios, along with zero files. Removing these two types halves the scores of dzip and bzip2 but leaves xz untouched. Xz still has 200 wins, now followed by gzip with 73, bzip2 with 41, and finally dzip with 36.

After eliminating the artificial files, the only data type that xz does not win or do well is YouTube data, a surprising result, given xz's compression power. However, it is important to recall from Section 8.4 that the CRs for YouTube data are extremely poor, even for the winners, so this is not a significant win for gzip in terms of effect. Ultimately, even though dzip is always close to the winner, it is very rarely the best choice. More importantly, xz is usually the winner. This suggests that rather than using complicated and time-expensive Methods like dzip, someone interested in maximum compression should just use xz.

8.8 Effect of Input Size

The previous experiments in this section occurred using 1M concatenated samples of the different data classes. However, some compressors, such as xz, will use extremely large windows (the portion of the file searched for matches) in an attempt to trade memory and CPU time for improved compression. This uses a significant amount of RAM. For example, as of this writing, xz will use between 325 and 1,300 megabytes during compression with xz -9, and will require 65 megabytes of RAM for decompression [112]. If such an approach could improve compression results for certain Methods and input data, limiting the input to one megabyte samples could significantly and artificially reduce the amount of compression achievable by those Methods.

Furthermore, this investigation is valuable because it teases out differences between dzip's performance and the real-world compressors like xz that are dzip's "competition." This is because even if some compressors are able to benefit from a larger amount of input in a dedicated, single-Method compression, dzip will not benefit because its largest input chunk size is 512K. While 512K is a significantly larger quantization chunk than is used in many AC systems, the result of this investigation may help indicate whether even larger chunk sizes are warranted.

To see if this ability would achieve greater compression on my sample data, I created 50-megabyte test sets for each data type by concatenating the relevant one-megabyte samples together. I then compressed each 50-megabyte sample set with dzip and the other compression Methods as in the previous experiments. Since the general effect is similar for all compressible data types, I describe the results of these tests for the datasets Wikipedia, Facebook, YouTube, and urandom below in Figure 27 through Figure 30. In these charts, the values for 1M samples have CIs (as in Section 8.4) because they are based on 50 samples each. The values for the 50-megabyte inputs do not have confidence intervals because there is only one large sample and CR does not vary across separate runs.

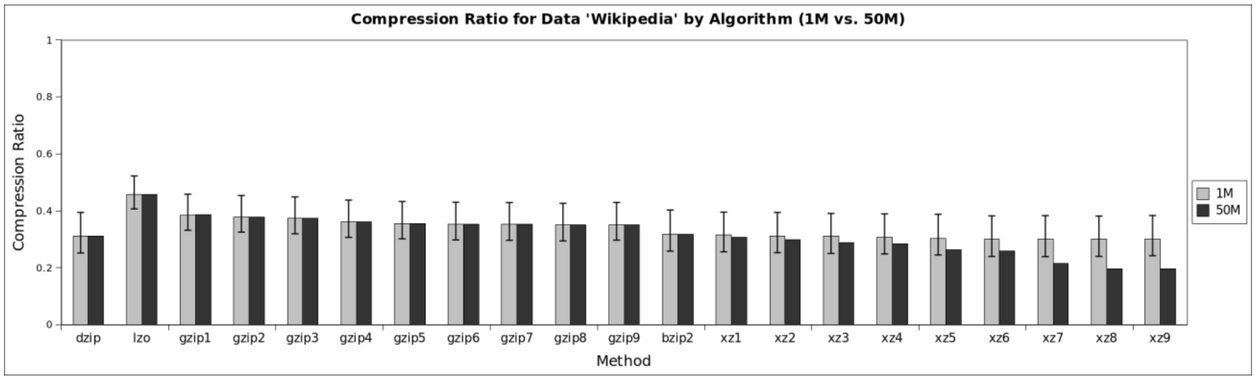


Figure 27. Comparison of average CR for Wikipedia data over 50 1M samples vs. one 50M file.

Figure 27 shows the difference between the CRs achieved by dzip (and the other compressors) for 50 one megabyte data samples and the single CR achieved by compressing all 50 megabytes of data as a single input. Looking left to right the CRs do not appear to diverge at all until bzip2, and the differences do not become statistically significant until xz -7 through xz -9. For these highest xz strength levels, the 50M tests compress the input an additional 10% (1M tests result in an mean CR of ~0.30 vs. ~0.20 for the 50M tests). As predicted, dzip is unable to benefit from xz's ability, because the largest input size that Datacomp considers at once is 512K.

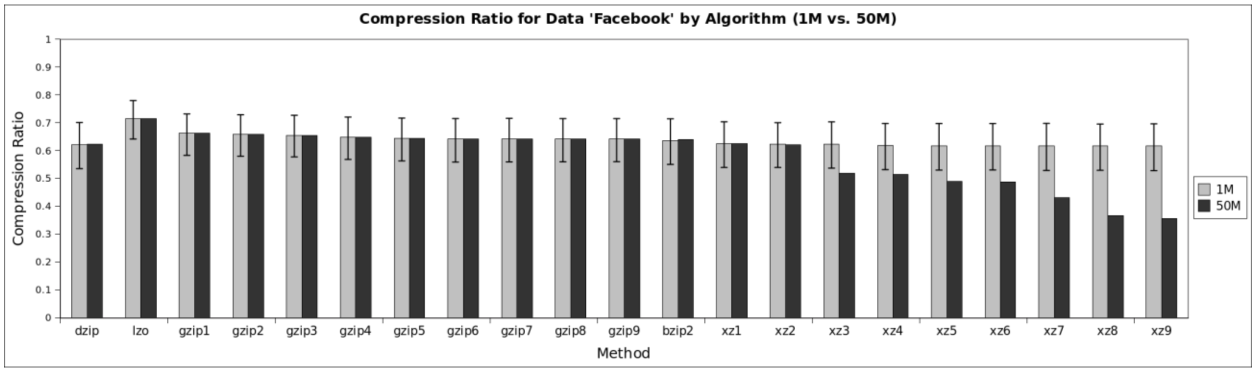


Figure 28. Comparison of average CR for Facebook data over 50 1M samples vs. one 50M file.

Performing the same experiments for Facebook data result in a similar, but greater, effect. Differences begin to be statistically significant for xz -3, where the 50M tests compress about 10% more, and increase up to xz -9, which compresses the input a further 26%. Interestingly, bzip2 actually performs slightly worse on our single 50M sample, although the difference is less than 1% and is not statistically significant.

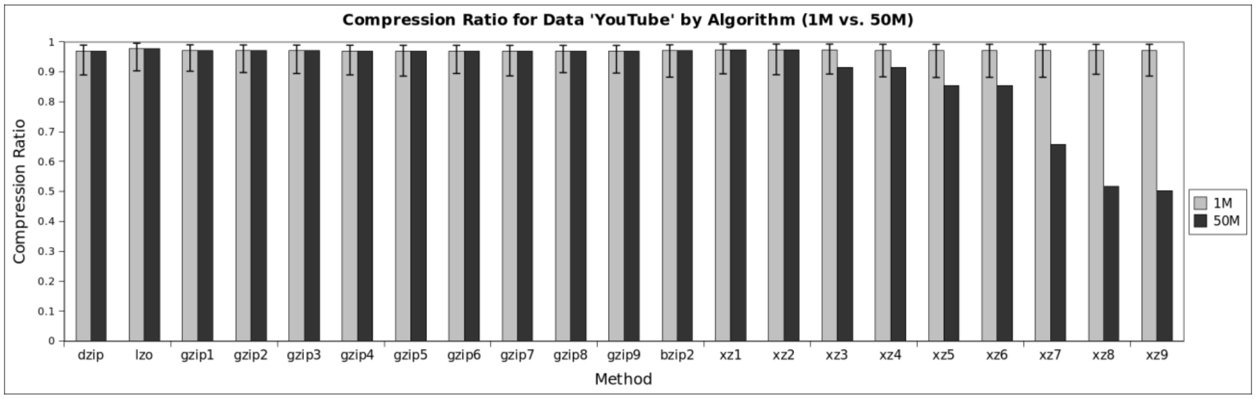


Figure 29. Comparison of average CR for YouTube data over 50 1M samples vs. one 50M file.

The ability of xz to improve compression on large input is quite remarkable on our YouTube data, as shown in Figure 29. The differences start being significant at xz -5, with a ~12% improvement over dzip, and grow to an astounding ~47% for xz -9. The end result is that while for 50 one megabyte samples xz9 only achieves a mean CR of 0.972, for one 50 megabyte input, xz achieves a CR of 0.503. This effect is not visible at all for other compressors, including xz -1 and -2, bzip2, LZO, gzip, or of course, dzip. Because of this astounding result, I manually repeated the experiment and obtained the same results.

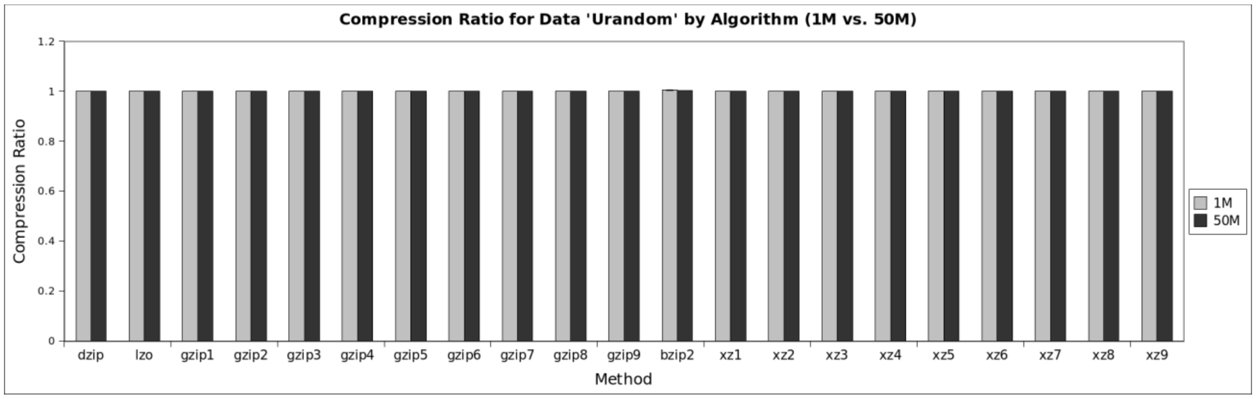


Figure 30. Comparison of average CR for urandom data over 50 1M samples vs. one 50M file.

Because of the tremendous results of xz on YouTube data, I thought it would be worthwhile to perform the same test on urandom data since it should not be compressible at all, regardless of how large xz's window size is. Figure 30 shows the results of this experiment. As expected, no compression is achieved for any algorithm on either 1M or 50M sample data.

In summary, xz clearly has a significant and almost unbelievable advantage if input sizes are large enough and the data admits compression. This suggests that even larger quantization sizes might be valuable for Datacomp (especially for xz), although larger sizes would increase the already significant cost of dzip's greedy search mechanism. In reality, it primarily serves to underscore the result that xz is a more effective and less costly strategy for maximizing compression than dzip.

8.9 Conclusion

Dzip almost always achieves its core goal of always compressing as well as the best static compressor. It also turns out that no static Method is best for all types of data. However, dzip is unable to meet its bonus goal of out-compressing static strategies through the use of quantization, even for one megabyte samples. Instead, the results suggest that while different compression Methods are indeed best for different data types, it is not the case that a mélange of compression Methods performs better than the best single Method. This is disappointing, but is nevertheless an interesting and useful result for AC.

Additionally, even though dzip generally meets our core goal of performing close to the best strategy on our main tests, it turns out that xz is typically the best static choice for data or similarly close to the best strategy. Furthermore, even though xz is "slow" and "heavy duty" compressor, it takes a fraction of the time required by dzip to complete its exhaustive compression operations, even at the highest strength level. Additionally, xz can significantly improve compression ratios when larger inputs are available and amenable to compression, something that dzip cannot do because of its current quantization approach and most compressors cannot do because of their window sizes. As a result, the dzip evaluation suggests that when attempting to maximize compression ratio, the best static strategy may be the best general compression mechanism available. From the selected set of compression Methods, that algorithm is xz.

However, turning this around, algorithms like gzip-6 are typically much faster and often statistically similar in terms of compression ratio than expensive algorithms like xz, at least when using input of the chunk sizes tested here. In these cases, middle-of-the-road Methods provide virtually the same space benefits at a fraction of the cost. Increased chunk sizes would enable xz to perform more to its potential (and out of reach of algorithms like gzip), but in reality sometimes large inputs are not available – the input simply is not large enough. An interesting direction for future work might be a more subjective adaptive Model that attempts to maximize space savings within some time budget – estimating whether faster compression is "good enough" for a circumstance relative to the cost of a compressor like xz.

9 COMPTOOL THROUGHPUT ANALYSIS

Comptool is a tool for performing exhaustive search in order to identify the best greedy AC Strategies for time, space, and energy. To do this, Comptool controls the output bandwidth, CPU load, and CPU frequency, quantizes data into chunks of various sizes, compresses them and finally “transmits” them over a throttled communication channel (as described in Section 5.3.3). Analysis software then uses a greedy approximation to identify the “best” strategy for the scenario in question. By performing a large number of individual experiments on similar opportunities, Comptool can identify strategies that are likely to have the intended effect in the real world.[78]

Because of Comptool’s exhaustive, discrete, and quantized approach for identifying compression Strategies, it generates a significant amount of information about the effects of combining a wide array of various compression parameters, all of which is stored in an SQL database. As a result, in addition to generating greedy best Strategies useful for validating or evaluating tools like Datacomp, Comptool serves as an “AC laboratory” providing useful information for researchers and developers.

In this section, I will discuss results produced by Comptool not already covered under Dzip Results (Section 8). These results will primarily be concerned with determining the best AC strategies for time and energy. Unlike the best compression strategy for space, the best time strategy is dependent on many variables. As a result, I will go into some detail regarding other compression parameters.

[78] Data and other information regarding these experiments will be available at http://labs.tastytronic.net/datacomp/

9.1 Experiment Design

While I compared dzip results to best compression strategies discovered by Comptool, the bulk of the Comptool testing for this research was in the area of optimizing throughput. These tests are similar to those for optimizing compression in the sense that the same greedy strategy is used to search for the "best" strategy given a scenario. However, where the compression ratio is dependent only on the data and the Method (algorithm, strength and chunk size), the compressed throughput is dependent on compression ratio and any execution environment factor that could affect the compression or transmission rates. With these latter categories, I performed tests while varying the bandwidth, CPU load and CPU frequency (at the levels described in the Workload section).[79] Because there is a large number of combinations of these factors, performing Comptool tests to maximize Effective Output Rate (EOR) is considerably more involved than testing for compression ratio.

Performing these experiments involves setting up environmental parameters as described in the Workload Design section, and then executing Comptool to identify the best strategies. Unlike dzip, where I concatenated the files in the sample set into a one megabyte block of data, for these experiments (and those in the rest of this section) each sample set contained the files as they were originally sampled using the method described in Section 7. Most sample sets contain only one file, although some contain more. This is relevant for compression throughput because the natural breaks in files limit the potential chunk sizes available. If one input file in a sample set is less than 256K, for example, then the 512K chunk size will not be used for compression.

Two test platforms were used for this work. The first is the LEAP measurement system, which is named *leap6*. This machine is an Intel Core 2-based system, with a quad-core (Q660)

[79] These are the parameters explored by Comptool, though no doubt there are other potential factors that could contribute to compression throughput.

CPU capable of operating at 1.6 and 2.4GHz. I used this machine primarily for energy-related testing, and to investigate the effect of CPU frequency, since this machine only has two frequency states. On *leap6*, I performed a set of 25 experiments on all nine data types at 6Mbit/s, 100Mbit/s, and 1Gbit/s. LEAP instrumentation captured the energy consumption during the execution of those workloads.

I wanted to demonstrate the potential of AC on current hardware, so I tried to find a system similar to my personal laptop in terms of computing power. My laptop is based on the Intel i7-2620M, a mobile-optimized quad-core CPU from 2011. As a testing platform, I acquired a desktop system (referred to as *laserbay*) based on the Intel Core i7 920 from 2008 with normal maximum frequency of 2.66GHz (and a maximum "Turbo" frequency of 2.93GHz). This processor is also nominally a quad-core chip, although with hyperthreading enabled it can appear to have eight cores to the operating system [113]. By disabling hyperthreading in the BIOS, this system becomes relatively similar in terms of computational power to my personal laptop.

I performed Comptool experiments at the bandwidths 500Kbit/s, 1Mbit/s, 6Mbit/s, 100Mbit/s, and 1Gbit/s as previously specified in the discussion of workload. On *laserbay*, I performed experiments on 50 sample sets for all nine data types, testing three CPU contention levels (0, 2, and 4 CPU monopolizing processes) on bandwidths 6Mbit/s and up. For the lower bandwidth levels of 1Mbit/s and 500Kbit/s, I had to reduce the number of experiments.

Comptool test runs performed at low bandwidths, particularly 500Kbit/s and 1Mbit/s, take an extremely long time. This is because Comptool compresses and transmits the entire input many times, and 500Kbit/s and 1Mbit/s runs are 2,000 and 1,000 times slower than 1Gbit/s, respectively. To limit the testing time, I performed 500Kbit/s experiments on only Facebook, Wikipedia and YouTube data, using two contention levels (zero and two contention threads). These experiments were performed over 25 one-megabyte file sample sets. For 1Mbit/s runs, I performed tests on 25 sets of all nine data types with two contention levels.

9.2 Energy and Time

Saving energy through AC is an important goal of Datacomp. Datacomp can, in fact, save a significant amount of energy over static compression strategies. However, while LEAP is a core component of Comptool, Datacomp does not use LEAP, nor does it include separate "time" and "energy" Models. Instead, Datacomp uses its time Model as its energy Model and vice versa. This is because my experimental data strongly suggests that the *difference* between the greedy best time and energy strategies (as determined by the current Comptool design and workloads) is less than the cost of using LEAP instrumentation as one of Datacomp's monitor modules. Additionally, while LEAP technology provides incredible energy measurement detail, it is not available in commodity systems (nor is anything like it). As a result, it seemed appropriate to design Datacomp without requiring LEAP.

I must stress that this is not a failure of LEAP technology in any way, but rather speaks to the very close relationship between time and energy in modern computers, as well as the design of Comptool as a standalone tool. In this section, I will describe the costs of LEAP, the practical energy considerations of modern hardware, observed differences between time and energy, the Comptool design choices that affected this outcome, and potential future directions.

9.2.1 Cost of Energy Measurement

LEAP can be used in one of two modes; the original "offline mode," where data is collected at runtime and analyzed after the fact, and a new "online mode" (developed by Digvijay Singh during the same time period as the works described in this dissertation). Online LEAP [84] provides current energy measurement data that can be used to inform choices for live workloads. Offline mode is naturally the cheaper of the two modes and is estimated to cost approximately 1% more energy than the workload without LEAP sampling.

Offline mode can be this inexpensive because the "heavy lifting" of analyzing sample data and synchronizing it to events takes place after the measured workload. For offline mode, the major runtime work of sampling is borne by the sampler hardware itself; the host needs only to power the sampler, generate the synchronization signal, and periodically transfer sample data from the sampler to a ramdisk. In contrast, online LEAP requires approximately 5% more energy than the workload without LEAP instrumentation [84]. This added overhead is incurred because online LEAP must do additional computation during the execution of the workload – synchronizing, parsing, and performing computations on the live sample data.

One to five percent is an extremely small overhead and in many cases is significantly smaller than the general energy effects being measured. Comptool can use LEAP in offline mode during all experiments. This provides Comptool with highly accurate energy data at a low cost that can be contrasted with the timing results for the same operations. If the differences are statistically significant, this could result in different "best" strategies for time and energy.[80]

However, Datacomp operates in real time and continually updates its history mechanism – it does not rely on fixed Models. If Comptool data suggested that the best energy strategy was sufficiently unique from other strategies to require its own energy Model, then Datacomp would need to run online LEAP as a "Monitor" for energy. Unlike Comptool, Datacomp must "make a profit" – the benefits gained by any AC component must, in the long run, outweigh its costs. In other words, to justify the use of online LEAP as a required component of Datacomp, Comptool's energy analysis would need to show a potential energy benefit for Datacomp of at least 5% over the "best time" strategy.

[80] LEAP instrumentation is optional, however, for people who do not have access to LEAP hardware.

Because an online LEAP Datacomp monitor would pay the overhead cost at all times, it would not be enough for Comptool to occasionally show a savings of more than 5%. To pay for online LEAP, Datacomp would need to save 5% more energy than it would save using a time strategy *on average* across all workloads using online LEAP. If the difference in benefits between the two systems is not large enough, then it makes sense for Datacomp to simply use the time Model as a proxy for energy.

9.2.2 The Race to Sleep

On most commodity computers, the *total* energy consumption of a computation workload by a system is strongly correlated to the runtime of the task, as opposed to the CPU frequency during execution or other factors affecting the energy use of individual components [114]. While many computers can control their CPU energy consumption in some coarsely-grained ways (e.g., raising or lowering the CPU frequency), the energy reduction comes with an attendant reduction in computational throughput, which increases the runtime for computational workloads. At the same time, other energy costs of the system (which are often proportionally larger than the CPU's energy use) do not vary with the CPU frequency. Instead, they are a static function of time if the computer is operational – they represent the costs "to keep the lights on." As a result, if reducing the CPU frequency (or some other change with a small impact) causes a task to take longer, the entire system will probably use more energy during the task execution, even if the CPU itself uses less.

At first, this makes it seem like CPU frequency variations (and other CPU energy-saving features) are pointless in the effort to save energy. This is not the case. However, it is important to make the distinction between hardware or software features enabling a computer to use less *total energy* over time, and those enabling the computer to perform a *specific computational task* while using less energy. In the first category are things like CPU frequency variation and replacing mechanical hard disks with solid-state disks, which can lower energy use overall. In the latter category are things like vector instructions (e.g., SSE) or hardware support for cryptography. Ultimately, on a thread-by-thread basis on the same hardware, reducing CPU frequency does not allow one to perform the same computationally-bound task for less total system energy.

Putting this together, if a computer is downloading data over a moderate network connection (e.g., Wi-Fi) and is not performing any other significant computational workload, then the CPU is mostly idle. In this case, it is cheaper in the long run to transition to a lower CPU frequency, because the CPU is not the current operational bottleneck – the network is. As a result, the CPU will use less energy but the task will take the same amount of time, resulting in a reduction in total energy used. On the other hand, if the data is compressed and the network is sufficiently fast, then decompression (i.e., computation) could be the bottleneck. In this case, operating the CPU at a lower frequency will simply extend the runtime of the decompression, incurring a greater total energy cost for the same task.

This basic property of most computer systems gives rise to what is known as the "race to sleep" or "race to idle" energy saving strategy in these situations. This heuristic is based on the idea that the most energy-efficient strategy is usually to perform the task as fast as possible (i.e., at the highest CPU frequency) and then transition into the lowest possible state. In this way, the computational workload is completed at the maximum speed, and then the computer can "sleep" or "idle" using the minimum amount of power. If the computer is no longer needed after the task is complete, it could literally shut down or go into "suspend mode," using almost no energy.

The "race to sleep" problem is well known for most current hardware. It is *conceivable* that newer (or future) systems with lower static costs and greater energy flexibility might show a more significant difference between the "best time" and "best energy" strategies. For example, the fact that the Intel Core i7 processors can put cores into different frequency states [113] could impact the energy trade-offs of parallel compression in different environments, since the number of threads used might significantly impact the energy consumption of the CPU. However, if the total static costs of the system are too great, they will probably still overwhelm savings at the CPU level. Ultimately, the number of LEAP-instrumented systems is finite and commodity hardware does not include energy measurement facilities capable of performing LEAP-like measurement.[81] As a result, without instrumenting additional platforms it is impossible to perform LEAP-enabled Comptool experiments to answer questions about those systems, and it would still be true that the savings due to energy measurement would need to consistently exceed approximately 5% on average in order to justify their inclusion in Datacomp.

9.2.3 Analysis

To investigate the resource difference between the "best time" and "best energy" strategies, I ran Comptool on our LEAP testbed, using the standard set of 1M data samples in a variety of different environments (see Section 7). As described in Section 5.3.1, LEAP instrumentation for Comptool (with the current experimental design parameters) significantly increases the time cost of Comptool experiments because there are literally thousands of individually measured subcomponents which may have comparatively long test times (as opposed to running many short tasks).

[81] While the Smart Battery interface of mobile devices provides full system power information, it is of much lower granularity and is less consistent over time because it is tied to the physical characteristics of the battery, which can degrade.

Because of the time cost, I performed experiments at one higher bandwidth (100Mbit/s) and one lower bandwidth (6Mbit/s). At 100Mbit/s, I compared all nine data types at both CPU frequencies (1.6GHz and 2.4GHz) and with three levels of CPU contention (zero, two, and four bzip2 processes as described in Section 7.4.2.2). 6Mbit/s tests take much longer, so I compared only YouTube, Facebook and Wikipedia data samples. For these three types, I compared both CPU frequencies and two levels of contention (zero and two bzip2 processes). I then computed both the "best time" and "best energy" strategies for each of the 1M sample sets of each data type in each environment. Each combination of an input (the sample set) and the environmental conditions comprises an individual "opportunity." Thus, for each discrete opportunity of a given type (e.g., YouTube data in some environment), I separately computed the best strategy for time and the best strategy for energy in the specified environment. This resulted in two tables: a table of "time results" and of "energy results" for the same set of opportunities.

A strategy optimized for energy still has space and time costs, just as a strategy optimized for time has space and energy costs. By calculating the absolute difference between the results for each opportunity row in the "time" and "energy" tables and converting that difference to a percentage of the "best energy" costs, I was able to compute a table of proportional differences between the two strategies in the environment.

Each table consisted of between 19-25 rows (depending on the test), each row representing the absolute difference between the "best time" and "best energy" results for a single sample set as a percentage of the "best energy" cost. A zero in a resource column (i.e., size, time and energy) denoted "no difference" between the strategies. I then computed the mean and confidence intervals for each resource column (e.g., size, time and energy) across all opportunities using the bootstrap as described in Section 8.3.

In this way, the difference between the two strategies could be summarized as a mean percentage cost per resource, complete with confidence intervals. Because these are confidence intervals over the difference between two alternatives, CIs that span zero – that is, the lower CI is negative and the upper CI is positive – signify a difference that is not statistically significant. If the CIs do not intersect zero (because they are both positive or negative), then the difference is significant.

Resource	Mean	Lower CI	Upper CI
Size	0.022%	0.006%	0.056%
Time	0.01%	0.0%	0.03%
Joules	0.563%	0.445%	0.674%

Table 18. Differences between "best energy" and "best time" strategies for 100Mbit/s tests.

Taking the mean and CIs over all data types and environments at 100Mbit/s results in Table 18. Most importantly, the numbers show that the mean improvement in energy savings for the "energy" strategy over the "time" strategy is 0.563%. This is less than the 1% overhead of offline LEAP analysis and almost ten times less than the 5% improvement needed to justify Online LEAP as a runtime energy monitor for Datacomp. However, because these results are statistically significant, they show that "race to sleep" is not always the best approach for saving energy, in that there is a consistent (albeit small) advantage to the best energy strategy.

Looking at all the data by opportunity rows (not shown), the maximum mean difference observed for any opportunity was 1.227% (Facebook, no CPU contention, 1.6GHz), while the minimum difference was 0%. The only obvious pattern is that "zero" data is the only type to consistently have no significant difference between time and energy strategies for any of its opportunities. Removing that data (on the basis that it is somewhat artificial) and recomputing the means and CIs increases the energy improvement to 0.639%. Running the analysis for only

Facebook, YouTube and Wikipedia in the 6Mbit/s environment results in an energy improvement of 0.688%. Thus, it does not seem that this effect exists only for a particular bandwidth or type of data.

The results of the time and energy strategies for the Comptool workloads are extremely similar – well below the 5% overhead of online LEAP measurement. This does not diminish the importance of LEAP or the usefulness of LEAP combined with Comptool (e.g., being able to explore component level effects of compression choices). It also cannot speak to the differences available for other current or future architectures, except to show that a significant change in energy consumption must be made in order to reach a strategy difference of 5%. As a result, given that the strategy difference is so small, live energy measurement is not "free" and is not available widely, I believe it is acceptable and prudent to simply use the time strategy as a proxy for energy in Datacomp.[82]

9.2.4 Impact of Evaluation and Comptool Design

These "time versus energy" results determined by Comptool must also be understood in the context of Comptool's "standalone" design. Because Comptool does not actually use the network, these results do not incorporate the energy effect resulting from the interaction of different strategies with the network hardware. This is not to say that the difference would be overwhelmingly large, as this depends significantly on the portion of the system energy consumed by the network device. Nevertheless, it could be significant either at the hardware level (as shown in previous research [84]) or potentially at the system level.

[82] Another issue (for Datacomp) is that Online LEAP requires up to 3.8 seconds after an event to return energy data (as of this writing). While this is incredibly responsive compared to the standard offline mode, the processing delay could lead to prediction error for systems that make choices more frequently.

For example, wireless communication can be costly. As a result, many wireless devices have a special “sleep mode” to save power between transmissions. During sleep, data cannot be sent or received, but energy use is diminished. If the cost of using the device is high enough, an AC strategy distinct from the best time strategy might conceivably be able to reduce energy cost at the interface level by creating more opportunities for the radio to “sleep” – even if this extends the total runtime of the transmission task.[83]

As discussed, the current Comptool design does not actually use the network. While it could easily be changed to perform network sends to a peer machine (instead of throttled writes to a local ramdisk), this would raise some new design questions. For example, one reason that it is currently acceptable for Comptool to reassemble experimental events outside of their original order into a composite AC strategy is that the communication channel used by Comptool is about as free from interference as possible. If Comptool instead wrote to the network (especially a wireless network), one would need to be very careful to control for interference.

Additionally, the current Comptool design does not pay for network protocol overhead, such as setting up TCP connections and the like. If Comptool were to transmit data across the network, it would need to pay for network connection overhead. Individual connections for each compressed chunk would be extremely costly in terms of time and would be subject to buffering effects. However, sequences of chunks sent together as one TCP connection could have ordering and buffering effects that would make it questionable to “slice and dice” the individual events in the way that Comptool is based. As a result, it is not clear how best to design Comptool for such a task. These are design issues for future work.

[83] That said, Digvijay Singh’s DEEPcompress investigation showed that optimizing for network hardware energy actually increased total system energy use. [82]

9.3 Maximizing Throughput

In this research, I use Comptool and Datacomp to explore a wide array of environmental parameters, data types, and compression Methods. This is useful because it allows us to see what the best-case scenario performance looks like, use this later as a "yardstick" against which to compare drcp performance, and also to inspect the effects of the various factors. As it turns out, not all of the investigated factors are equally significant. In order to make sense of the many issues at play, I will start with the most significant factors – the environmental bandwidth, the input, and the available compression Methods, and will then drill down to lower level details including chunk size, CPU frequency, and CPU contention.

First, though, I will briefly review some of the discussion from Section 6 on the design of Datacomp, which illuminates how different compression algorithms end up being the "best" for a given opportunity. The key equation describes the Effective Output Rate (EOR) of a communication:

EOR = (uncompressed_size / operation_time)

This equation holds true whether or not the data is compressed as a part of *operation_time*. For the purposes of compressed communication, a slightly different equation is more useful:

EEOR = min(available_bandwidth, algorithm_output_rate) / compression_gain

… where the *compression_gain* is the inverse of the compression ratio (e.g., a compression ratio of 0.50 corresponding to a halving of size corresponds to a compression gain of 2.0 – a doubling of the data).

The EEOR equation is important because it captures the relationship between the importance of the compression ratio achieved for a given operation and the importance of the speed of the compressor. In combination with the available bandwidth of the network, these terms determine if and how a particular Method will be able to improve throughput.

Some basic rules of thumb help to shed light on how and why various algorithms succeed in different circumstances. The first rule of thumb is that **if a Method's output rate is *greater* than the available bandwidth, then *only* the compression ratio of the Method will impact the throughput – it does not matter *how much greater* the algorithm's output rate may be.** In this situation, the compressed communication rate will be *at most* the channel's bandwidth. After decompression, this rate is multiplied by the compression gain. Thus, given a set of algorithms that all have a greater output rate than the channel, the best choice is the algorithm with the greatest compression ratio.

The second rule of thumb is that **if a Method's output rate is *less* than the available bandwidth, then both the compression ratio and Method output rate will impact the throughput**. In this circumstance, the Method is not capable of "flooding" the network; that is, the network will be underutilized. The compression gain will still be applied to the data the Method is able to emit, which may enable an effective throughput greater than the available bandwidth or a different Method. However, compared to other Methods, an output rate deficiency is like a compression ratio *penalty* – a slower Method must have a proportionally greater compression gain to be competitive.

9.4 Data Type and Bandwidth

In this section, I will discuss the results of using Comptool to identify the "best" throughput strategies for different data type and bandwidth combinations. To save space, I will describe the details of the primary plot types before discussing specific results.

The first type of plot is exemplified by Figure 31. This type of plot shows the absolute mean time to compress and send a number of 1M binary sample sets using various Methods, limited by some bandwidth level. Lower is better. The first column, "null" shows the time cost of simply sending the entire data at once, uncompressed. The second column, "best," reflects the result of the "greedy best" strategy for time, and is composed of various quantized compression events selected from all possible Methods. Like "null," the results of the other compressors were obtained by performing compression on the entire (non-quantized) input at once before "sending" it. As with dzip, full compression was used to compare Comptool against standard compression mechanisms that are not impacted by quantization overhead. Confidence intervals for this plot (and every other plot in this section) were taken using the bootstrapping method described in Section 8.3.

The second type of plot is exemplified by Figure 32. It shows the improvement for a scenario produced by best, lzop, gzip-1, gzip-6 and xz-1 versus null as a percentage of the cost of using null compression.[84] Higher is better. These values were computed by taking the difference between each Method and null and dividing by the cost of null. In this way, a positive percentage shows improvement over null compression, while a negative percentage shows the cost of using the Method in question. I computed the difference between each Method and null using the same process for comparing Time and Energy differences as described in Section 9.2. Because the differences are computed against the cost of null compression, CIs that span zero are statistically indistinguishable from null compression.

Finally, the third and fourth types of plots in this section are exemplified by Figure 33 and Figure 34. They provide insight into the types of choices being made by Comptool by counting up number of times that particular compression Methods and discrete chunk sizes were

[84] Null is not pictured as it would always appear as 0%.

used as a part of the “best” Method for that Opportunity. The number is described as a percentage of the total number of Methods or chunk sizes used. This helps to illustrate the composition of different strategies for specific opportunities.

9.4.1 Wikipedia

Figure 31 shows the runtime of executing several Methods on Wikipedia data in addition to the effect of bandwidth on the runtime of the compressed transmission. Clockwise from top left, the bars show the elapsed time of compressed transmission over 1Mbit/s, 6Mbit/s, 1Gbit/s, and 100Mbit/s communication channels. As expected, 1M takes 8 and 0.008 seconds to complete with null compression at 1Mbit/s and 1Gbit/s, respectively. This reveals that null compression operations have little to no time overhead skewing the measured time. If Comptool was unable to produce output at one of these levels, such as 1Gbit/s, then more expensive Methods would also suffer from a time penalty and non-AC Method choices in those environments might not be accurate.

The next obvious result is that at all four bandwidth levels (and at 500Kbit/s, although this is not pictured), one or more strategies are capable of improving throughput (and thus time and energy) for this data over no compression. Also, unlike the results for compression ratio (see Section 8) where “best” was often slightly worse than the best static choice, the “EOR best” is more frequently the best (lowest) mean time, or is tied for best with another Method. That said, at lower bandwidths, many static strategies are statistically indistinguishable (or it is not clear whether the differences are significant).

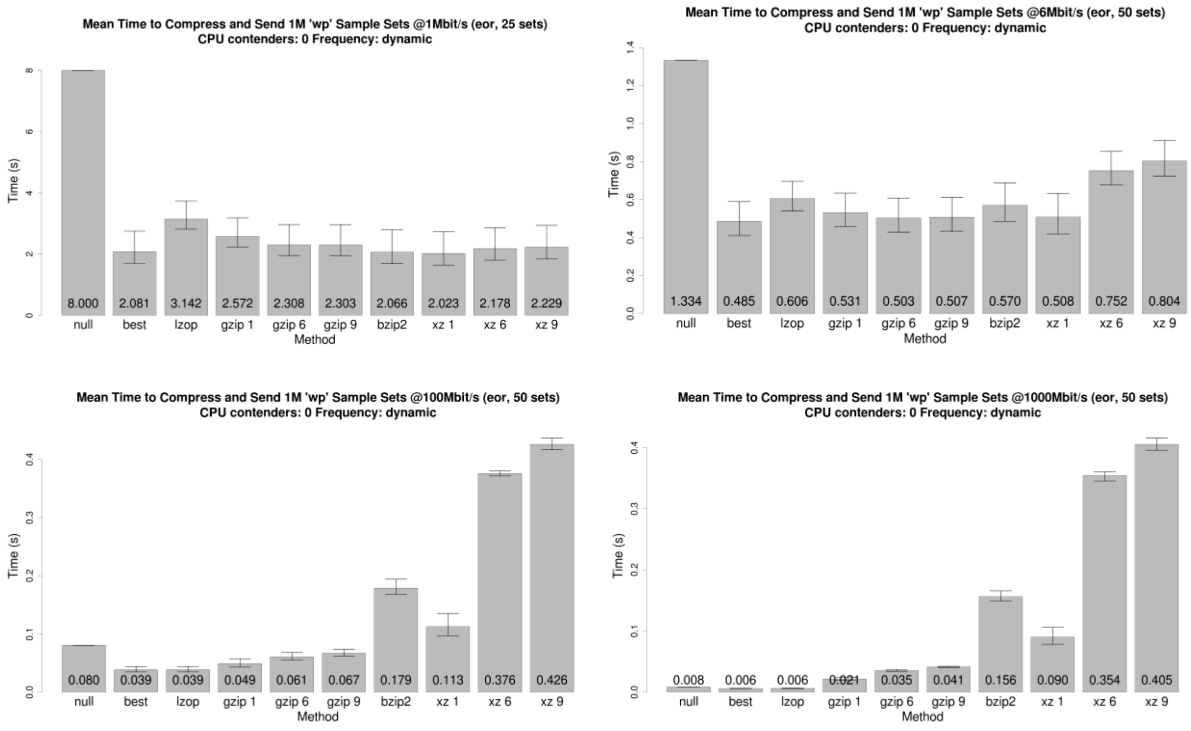


Figure 31. Combination time plot 1M-1G

This is directly related to our first rule of thumb as just described above. Because all the AOR for all Methods is greater than the ABW, and because the CRs for most Methods are statistically indistinguishable, there is little statistical significance between the various Methods. The one exception is that there is a statistically significant difference between “best” and lzop. This is still a function of CR, however. Lzop is a tremendously fast compressor with medium to poor compression. In this scenario, its speed does not help it, while its poor compression hurts it. As bandwidths increase, the differences between the algorithms become more distinct, and by 100Mbit/s, most Methods are statistically different from one another. This is a function of our second rule of thumb; as bandwidths increase above the AOR of several Methods, the combination of their CR and AOR for the given opportunity becomes more important. As a result, the utility of slower Methods decreases (e.g., xz-9 at 6Mbit/s) and then becomes a significant penalty (e.g., xz-9 at 100Mbit/s and 1Gbit/s). In the end, lzop, which was the worst static Method at 1Mbit/s, becomes the best static Method at 1Gbit/s. Similarly, xz-1, the best static Method at 1Mbit/s, becomes more than 10 times worse than no compression at 1Gbit/s.

Another way we can contrast the benefits of AC to existing static strategies is shown in Figure 32, which depicts the percentage differences of a subset of the previous Methods. This helps put the differences in perspective. We can see, for example, that best's improvement over no compression at 1Mbit/s is nearly 74%. It also highlights the fact that xz-1 has a mean time just slightly better than "best." Differences decrease at 6Mbit/s, primarily because the increased bandwidth improves lzop's performance in comparison. However, at 100Mbit/s, xz-1 has become a loss while gzip-1 and gzip-6 (the default gzip strength) falter. At the same time, "best" and lzop are clearly the best strategies and are still able to improve performance over null by more than 50%. At 1Gbit/s, this improvement is cut down to ~28%, but no other Method is able to improve runtime at all, with the third place Method, gzip-1, costing nearly 161% more than not compressing.

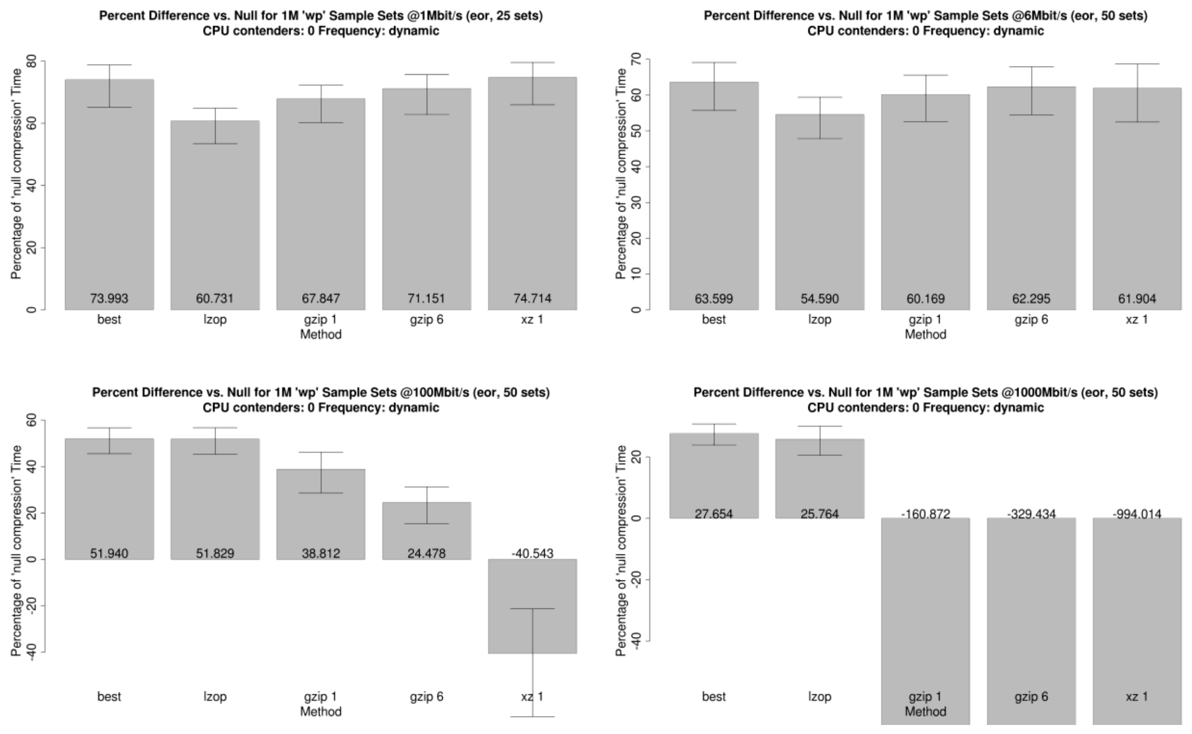


Figure 32. Percent difference in mean time for Wikipedia data at various bandwidths.

Still another way to examine differences in Method choice as bandwidth increases is to count the number of times a specific compression Method is selected for a given opportunity. Figure 33 shows these counts, including for experiments run at 500Kbit/s. Each bar represents the total number of times a particular Method was chosen as a percentage of the total number of choices made in that opportunity.[85]

[85] There are no confidence intervals to display because these are totals and not means.

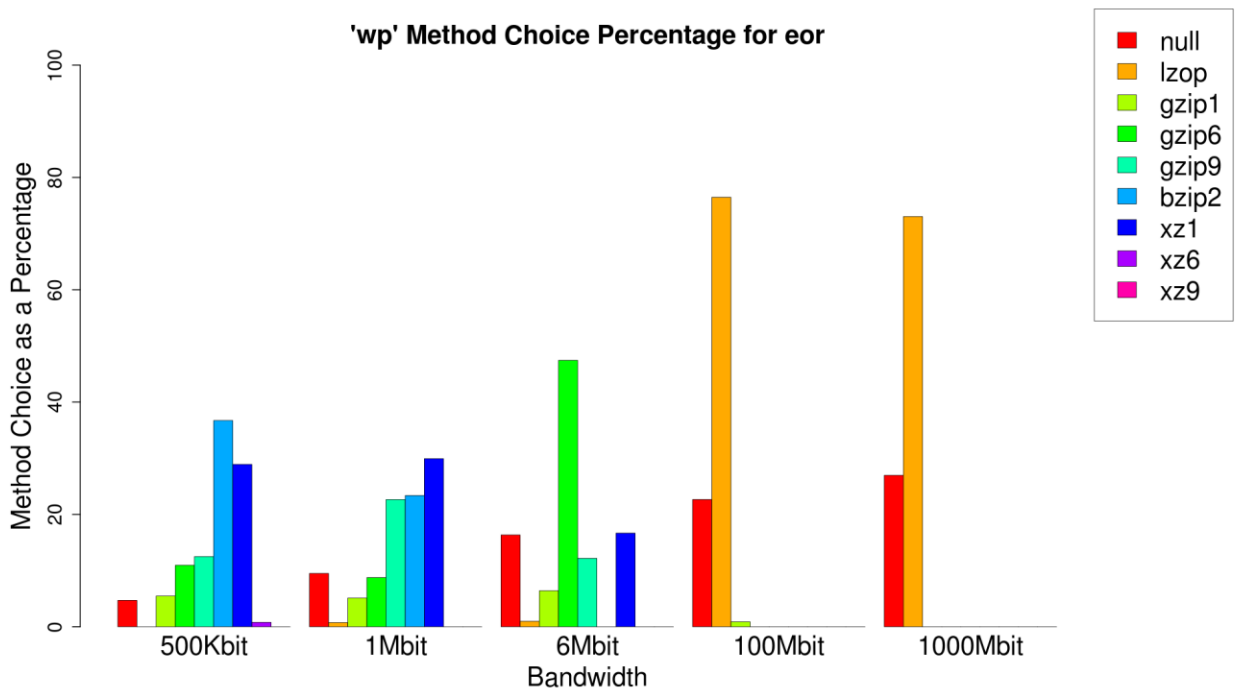


Figure 33. Method choice for Wikipedia data at multiple bandwidths.

I hypothesized that Wikipedia data would be largely compressible, and this appears to be so, given the previous time results and the small number of null compressions at lower bandwidths. While the choices for 500Kbit/s and 1Mbit/s are similar, the stronger Methods are more heavily represented at the lower bandwidth. However, the improvement results in the previous figures are not significantly different because the CRs for the stronger Methods are not that much better. This could be a result of 512K chunk sizes being too small, as is suggested by the dzip results, so in future work it would be interesting to test whether performance would improve with some "jumbo" chunk sizes.

Other trends are visible. By 6Mbit/s, bzip2 has been completely eliminated in favor of gzip6, although xz-1 is still the second most common choice. As one would expect, as bandwidth increases, most compressors cease to be profitable (due to our second rule of thumb), and as a result the proportion of null and lzop choices increase. At higher bandwidths, including 1Gbit/s, lzop (which was never chosen at 500Kbit/s and rarely chosen at 1 and 6Mbit/s) dominates the choices. This serves to show that the best strategies (for a given Opportunity) are indeed made up of a selection of various compression Methods and that the proportions of those strategies adapt with the changing bandwidth.

Finally, Comptool was built in part to explore the notion that making the best compression choices could depend on choosing to compress the right *amount* of data at each opportunity in addition to choosing the best Method. To test this hypothesis, Comptool and Datacomp explore the use of multiple quantization levels ("chunk sizes"). To visualize the effect of using different chunk sizes, I plotted the chunk size choices the same way that I plotted Method choices.

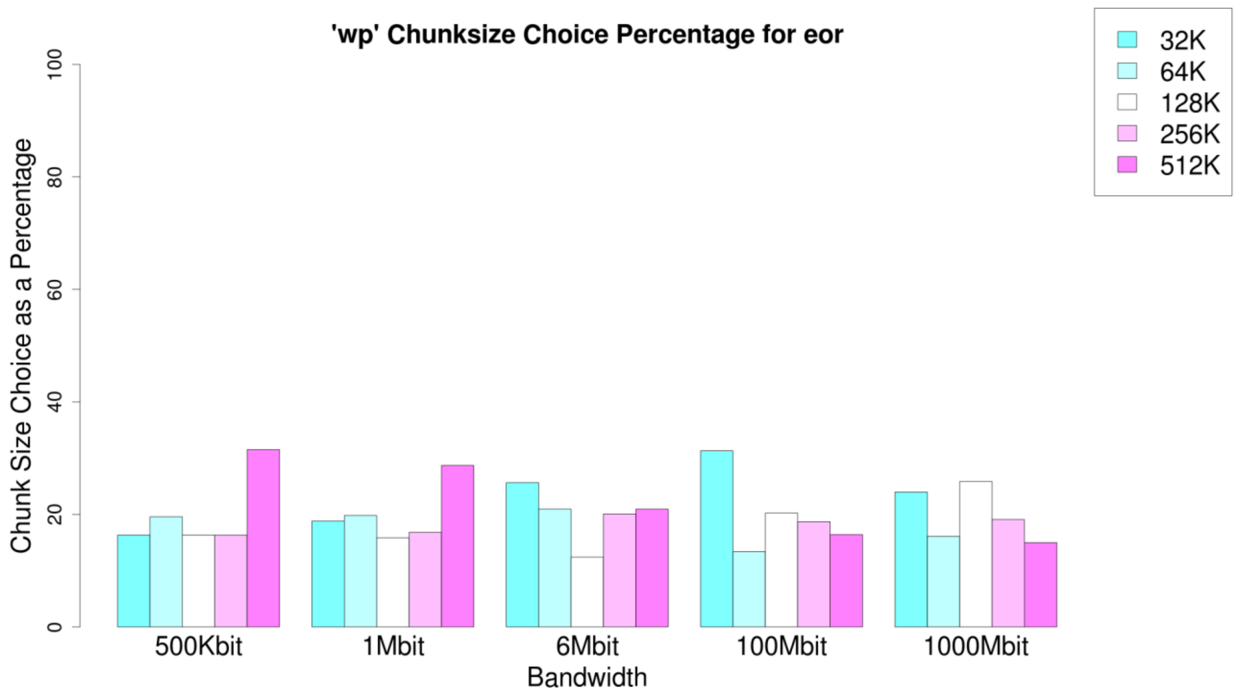


Figure 34. Quantization level choice for Wikipedia data at multiple bandwidths.

The quantization levels selected by Comptool to optimize Wikipedia data for EOR are shown in Figure 34. While the differences between each bandwidth are not especially large, one clear trend is a reliance on 512K chunks at the lowest bandwidth, which steadily decreases until by 1Gbit/s it is the least chosen chunk size. At the same time, 32K chunks are in the smallest group at 500Kbit/s and increase up to 100Mbit/s. Then at 1Gbit/s, 128K chunks take the lead and 32K chunks take second place. A conclusive answer as to why this occurs is beyond the scope of this research. However, this data suggests that different chunk sizes are important at different bandwidths.

9.4.2 Facebook

Facebook data is also surprisingly amenable to compression. As shown in Figure 35, all Methods improve the throughput at 1Mbit/s and 6Mbit/s (and at 500Kbit/s, not shown). The mean runtime for "best" (5.316s) is again better than the mean of all other techniques, and in this case is closest to xz-1 (5.368s), although the differences are not significant at 1Mbit/s. At 6Mbit/s, there are some significant differences between the slowest and fastest compressors, but it is not until 100Mbit/s and up that the differences become more distinct.

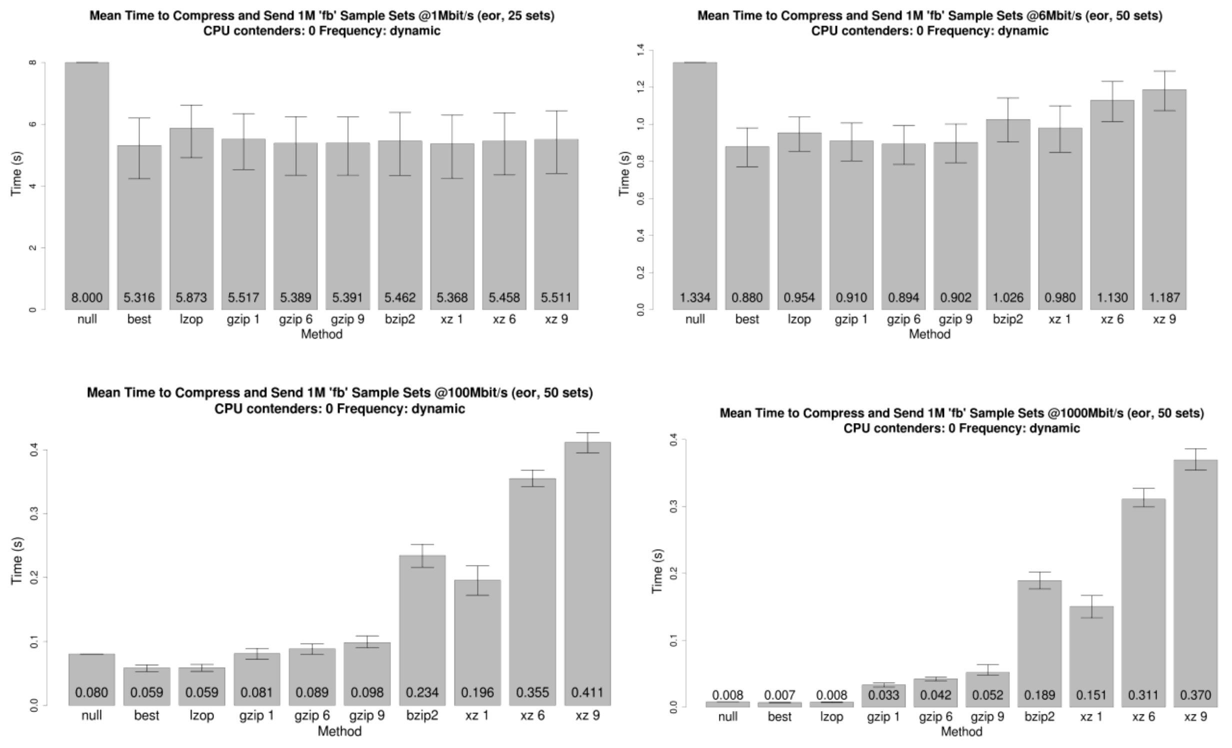


Figure 35. Mean time to send 1M Facebook data at various bandwidths.

At 100Mbit/s, best and lzop have identical performance and still significantly improve upon null compression. However, gzip-1's improvement is not statistically significant and all other Methods fare worse, including bzip2 and the xzs, which were able to improve throughput at the lower bandwidths but are wasteful at and above this rate. At 1Gbit/s, these effects are even more pronounced. The "best" strategy is still an improvement upon null, although lzop is now statistically indistinguishable from null compression. Of note is that at both higher bandwidths, xz-1's mean performance is considerably better than bzip2's (although this difference is only statistically significant at 1Gbit/s).

The percentage of effect (as shown in Figure 36) highlights that "best" always has the greatest mean effect improving throughput by 33-34% at 1 and 6Mbit/s, but that the difference between best and the other Methods is not statistically significant. At 100Mbit/s, best and lzop are both a significant improvement over null, saving ~27%. The mean result for gzip-1 and gzip-

6 are now statistically indistinguishable from null, while xz-1 is now a clear loss. At 1Gbit/s, best is still able to save 14% and lzop 8%,[86] but the other Methods are significant losses.

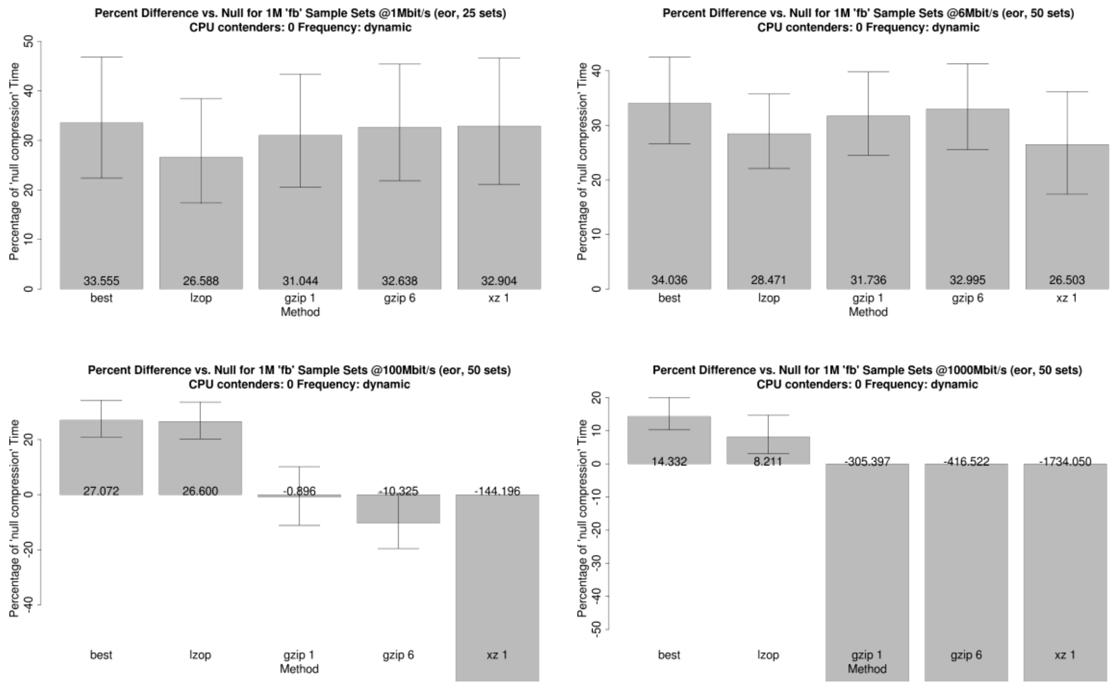


Figure 36. Percent difference in mean time by algorithm to compress and send Facebook data.

The fact that "best" is able to improve its mean beyond lzop's suggests that the best strategy for Facebook data at 1Gbit/s uses a combination of null and lzop, and that a "pure lzop" strategy was no longer profitable. The Method choice data in Figure 37 supports this notion, as lzop is the main Method chosen at 100Mbit/s, while at 1Gbit/s it is second to null. As with Wikipedia Method choices (Figure 33), stronger algorithms are selected at lower bandwidths, which transition (primarily) to gzip at middle bandwidths, and finally to lzop and null at higher bandwidths. Also similar to Wikipedia, the proportion of chunks compressed using null increases with the bandwidth.

[86] Both of these improvements are statistically significant compared to null compression, but not with respect to each other.

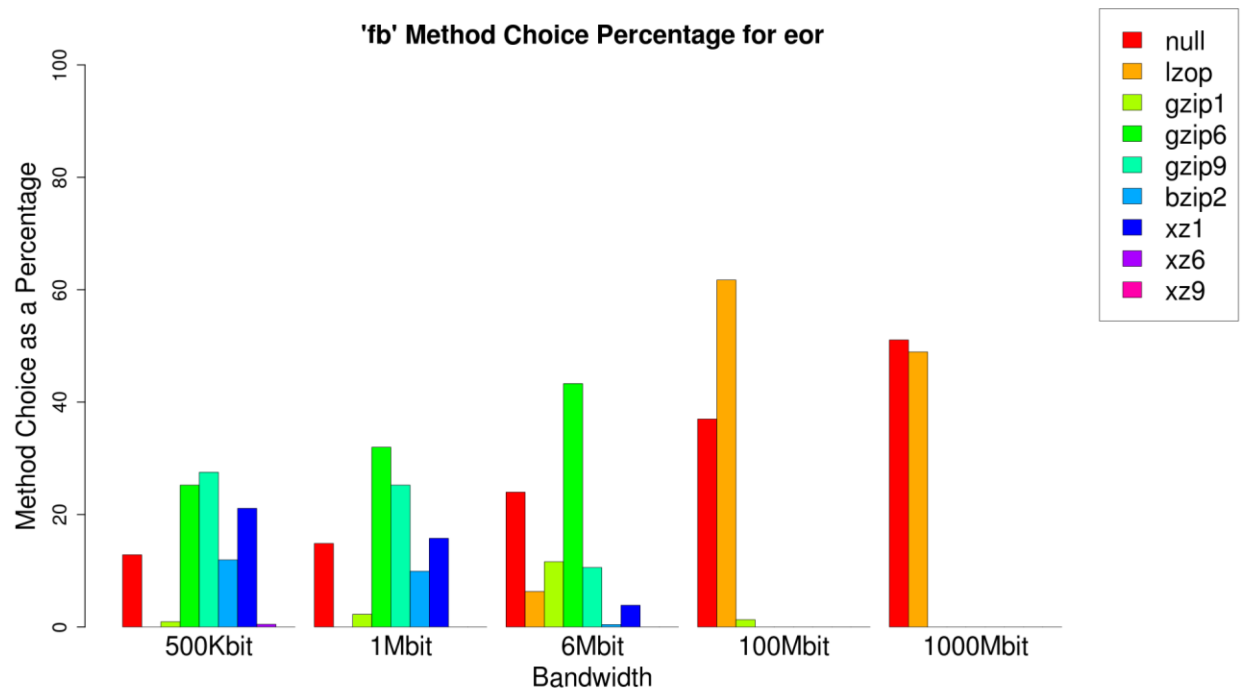


Figure 37. Method choice for Facebook data at multiple bandwidths.

The choice of chunk sizes for Facebook data (shown in Figure 38) do not show especially distinct patterns, except for a general increase in 32K chunks between 500Kbit/s and 6Mbit/s, which tapers off at 100Mbit/s. Between 6Mbit/s and 1Gbit/s there is a significant increase in 512K chunks; at 1Gbit/s there are approximately twice as many 512K chunks as at 1Mbit/s.

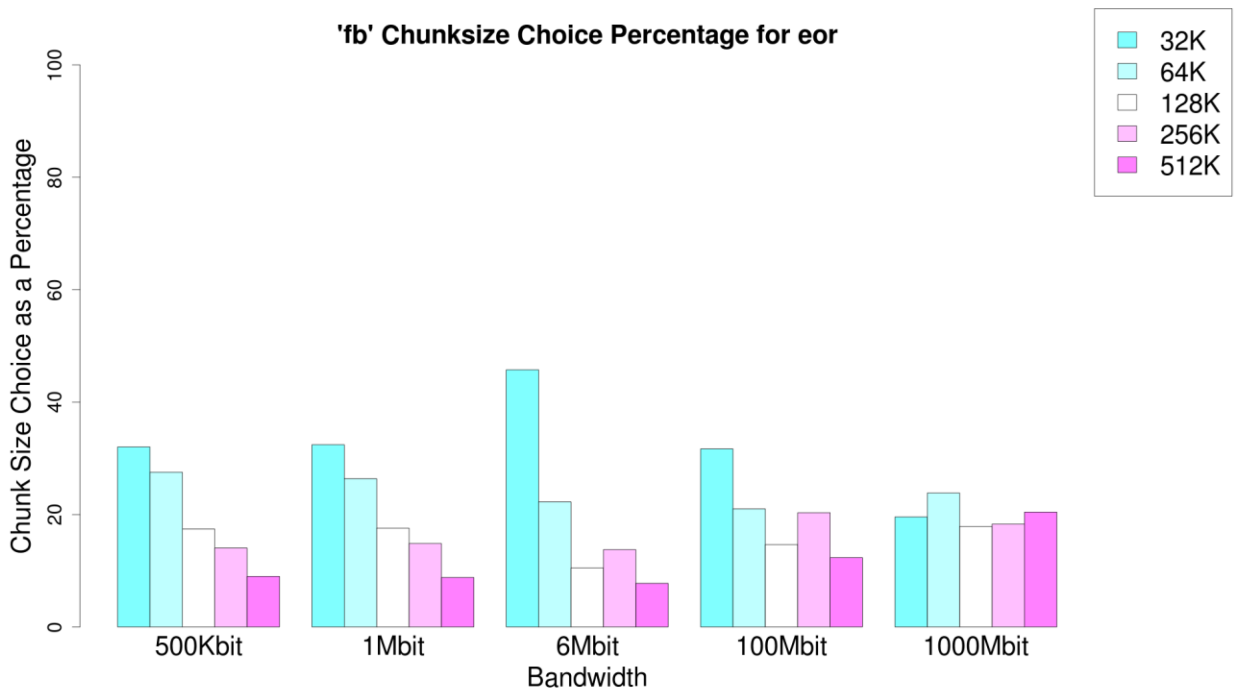


Figure 38. Quantization level choice for Wikipedia data at multiple bandwidths.

9.4.3 YouTube

The runtime results for YouTube data, shown in Figure 39, are the first example of data that is substantially uncompressible with the Methods used by Comptool. As a result, at 1Mbit/s the mean times for all non-null Methods are within approximately 0.3 seconds out of 8s for null compression. Two Methods, xz-6 and xz-9, are already significantly worse than null compression at 1Mbit/s. The "best" strategy is "significantly" better than null compression, saving about 0.1s per 1M input. All other Methods at 1Mbit/s are statistically similar to null.

At 6Mbit/s, the situation is similar (although lzop is actually just barely statistically significantly better than null). At both 100Mbit/s and 1Gbit/s, "best" is still enough better than null compression to be statistically significant, but the absolute differences are tiny – a 0.0017 second improvement over 0.08 at 100Mbit/s and a 0.0001 second improvement over 0.008 at 1Gbit/s. At 100Mbit/s, lzop is statistically indistinguishable from null, but at 1Gbit/s is finally statistically worse, losing 0.0005 seconds per 1M. The surprising result here is that *any* amount of compression can improve throughput of YouTube data at 1Gbit/s. I also find it surprising that lzop as a static strategy is almost able to keep up with this data even though it is not able to improve run time.

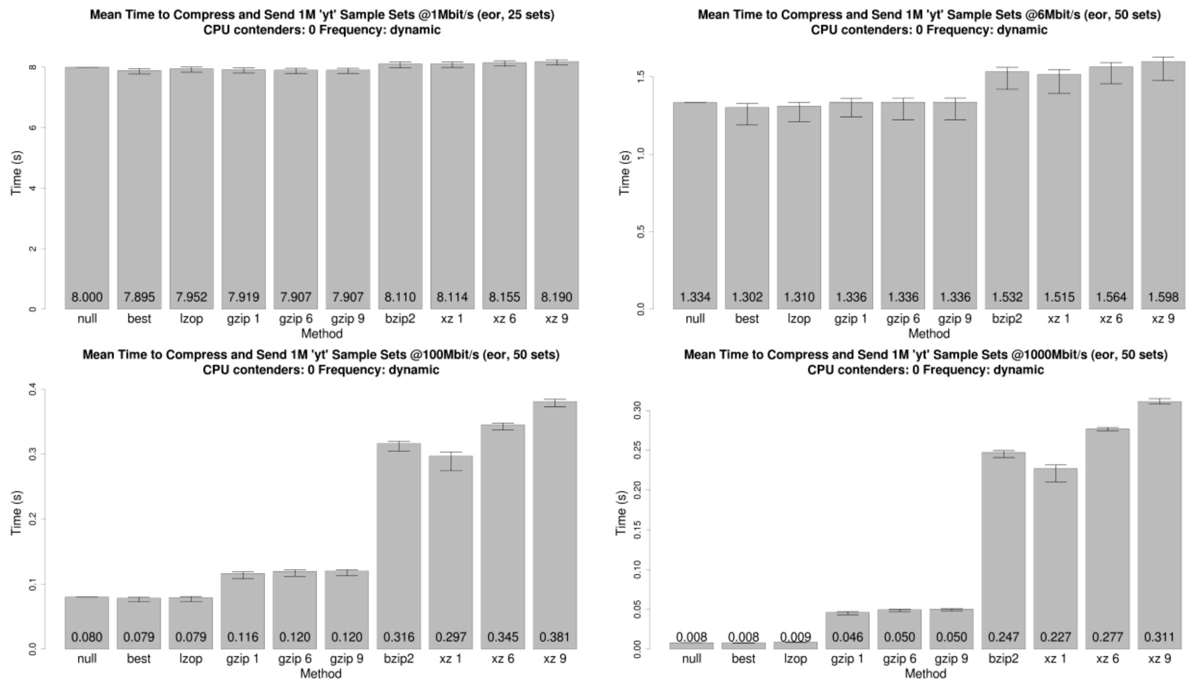


Figure 39. Mean time to send 1M YouTube data at various bandwidths.

The percent improvements for YouTube (shown in Figure 40) illustrate these differences more clearly.[87] In particular, they highlight that while the absolute time differences are tiny, the percentages are still statistically significant. Streaming video is a huge business and a significant portion of modern Internet traffic – if an AC mechanism could save even one or two percent it could be worthwhile to the stakeholders.

This highlights Comptool's use for developers and site designers not as an adaptive tool, but as a tool for profiling compression techniques for application data. Developers could use their application data as input for Comptool, identifying where savings could be found and

[87] Please note the different y-axes due to xz's increasing overhead.

building those techniques into their applications. While only certain compression techniques are included in the HTTP standard, sites relying on Flash video players or other client-side software could upgrade their software to benefit from the Methods identified by Comptool.

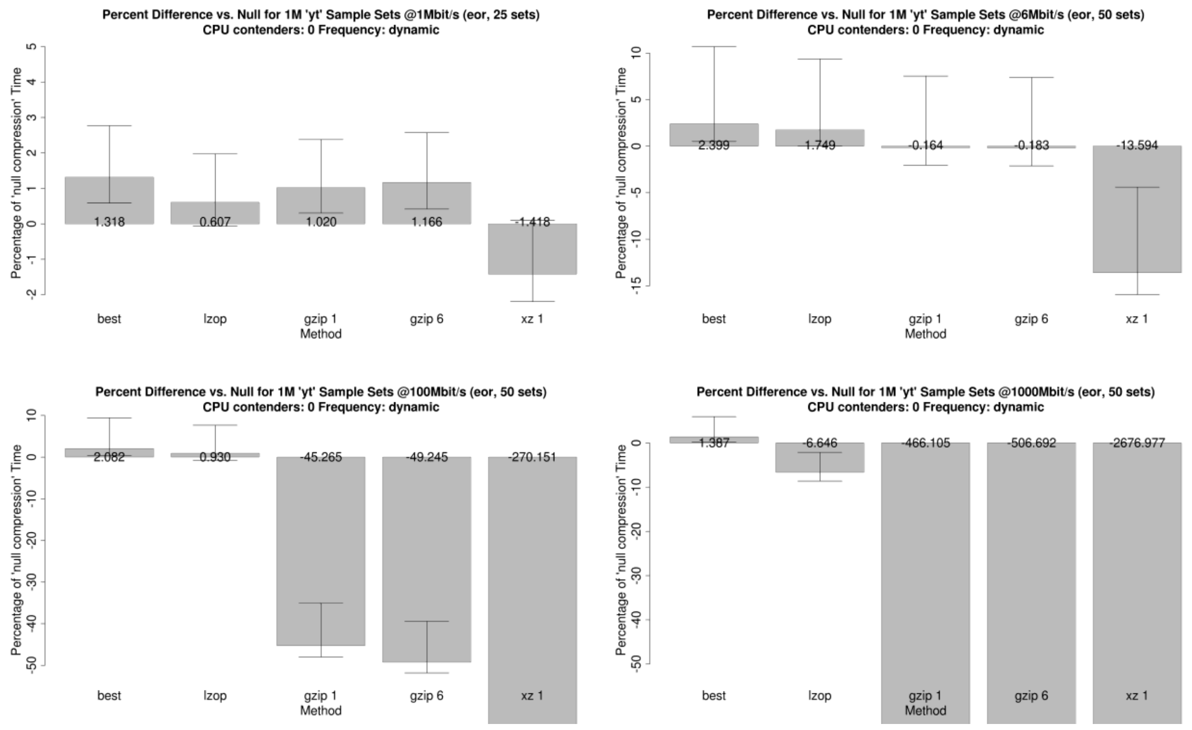


Figure 40. Percent difference in mean time by bandwidth for YouTube data.

Method choice percentages for YouTube data (shown in Figure 41) clearly illustrate the strategies selected by Comptool for YouTube data. At the lowest bandwidth (500Kbit/s in this chart), gzip dominates. This makes sense, as gzip was the most effective compressor for YouTube data in the dzip results (Section 8.4). However, as the bandwidth increases, null rapidly overtakes all other Methods and what little compression is performed is done by lzop.

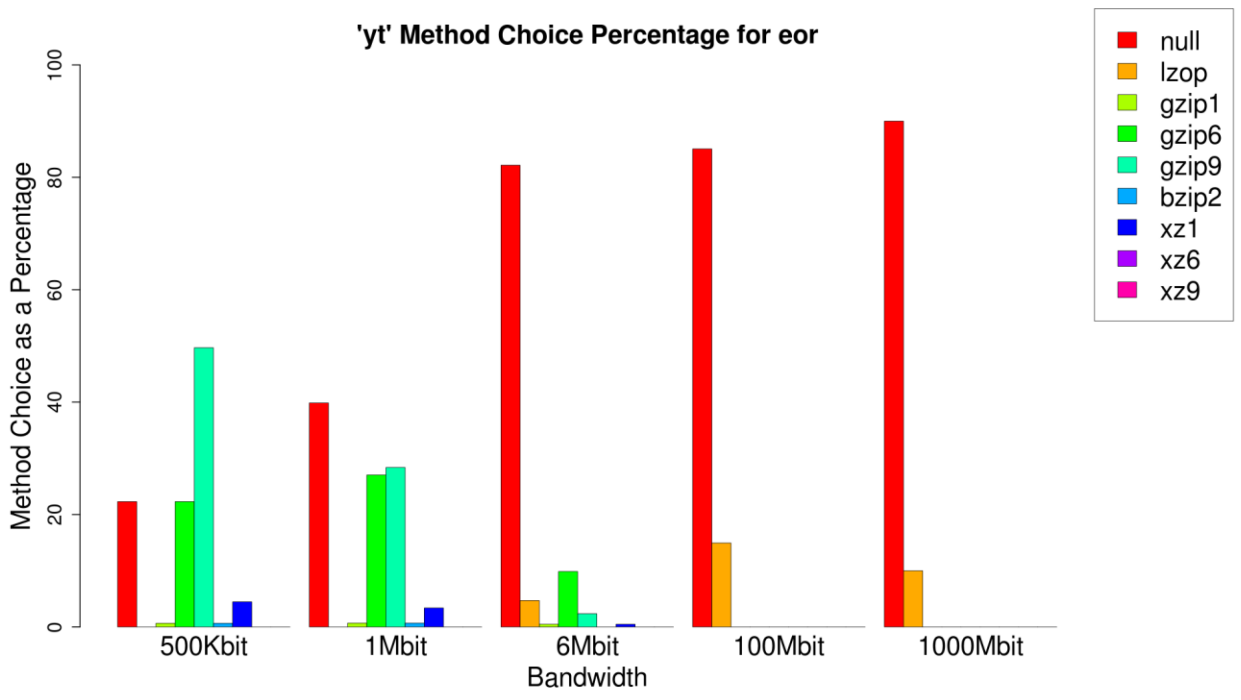


Figure 41. Method choice for YouTube data at multiple bandwidths.

Chunk size choices, shown in Figure 42, demonstrate a clear preference for smaller chunk sizes at lower bandwidths, and larger chunk sizes at higher bandwidths. Since the stronger Methods (bzip2 and xz) are not the key components of the low-bandwidth "best" strategy, it is not necessary to rely on large input chunk sizes. On the other hand, as most events are converted to null compression at higher bandwidths, there is less benefit to using smaller chunk sizes and greater risk of additional overhead due to quantizing.

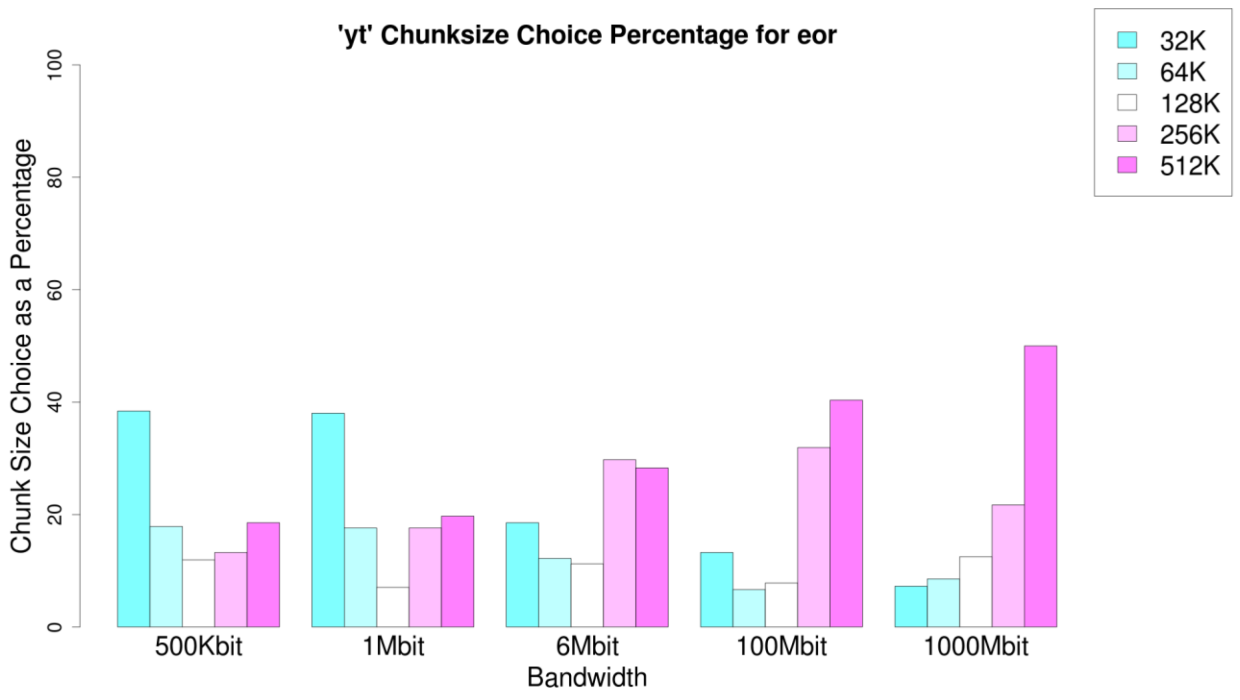


Figure 42. Quantization level choice for YouTube data at multiple bandwidths.

9.4.4 User Web Data

Perhaps the most interesting thing about the User Web data results (Figure 43 through Figure 45) is how similar they are to the Facebook results (Section 9.4.2), even though the volunteers who surfed the web were specifically asked not to use Facebook. Most statistics are very close between the two data types, although User Web data appears to be slightly harder to compress. I draw this conclusion based on the fact that the performance penalties for strong algorithms are somewhat higher for User Web data than Facebook, and because Method choice counts for User Web data shows a greater proportion of null compression used at 1Gbit/s when compared to Facebook data. Chunk size choices are not shown.

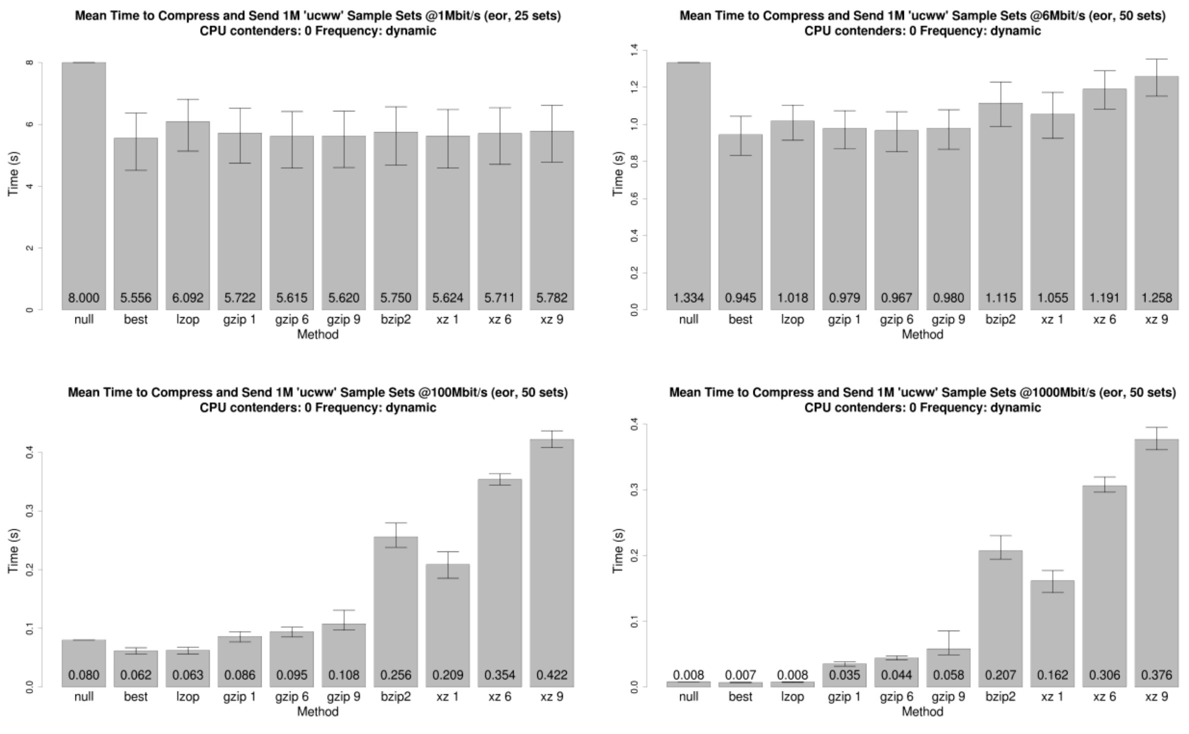


Figure 43. Mean time to send 1M User Web data at various bandwidths.

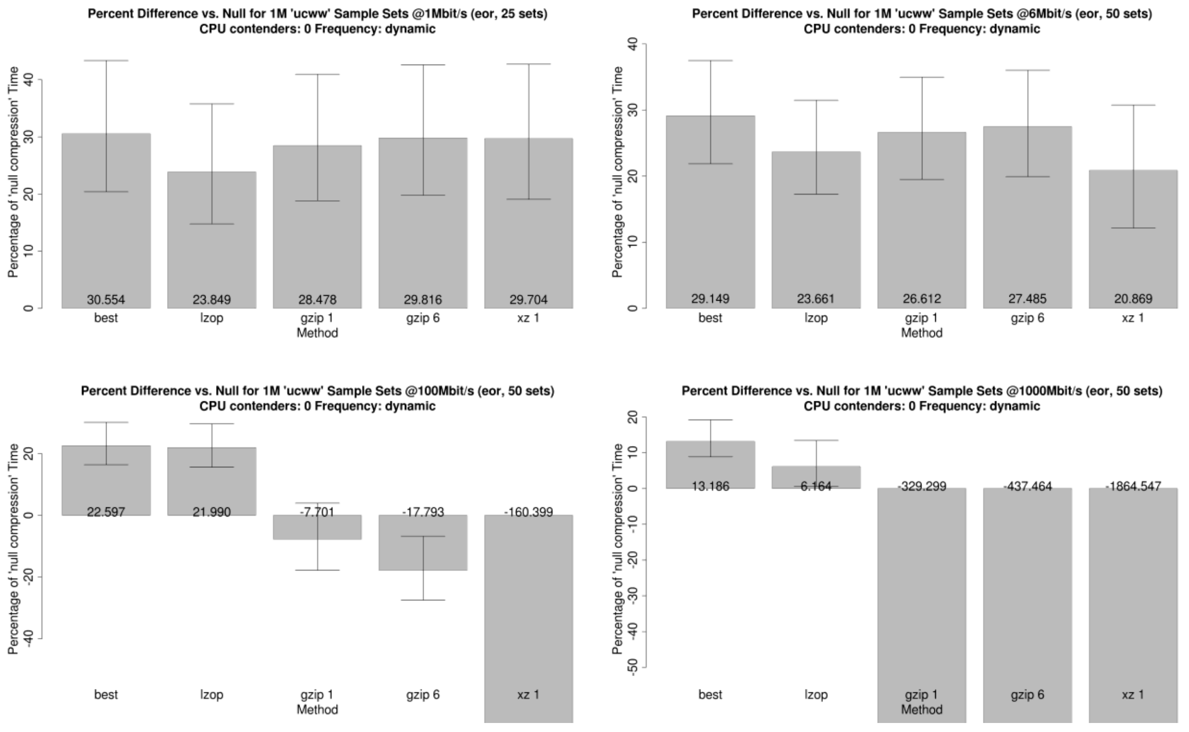


Figure 44. Percent difference in mean time by bandwidth for User Web data.

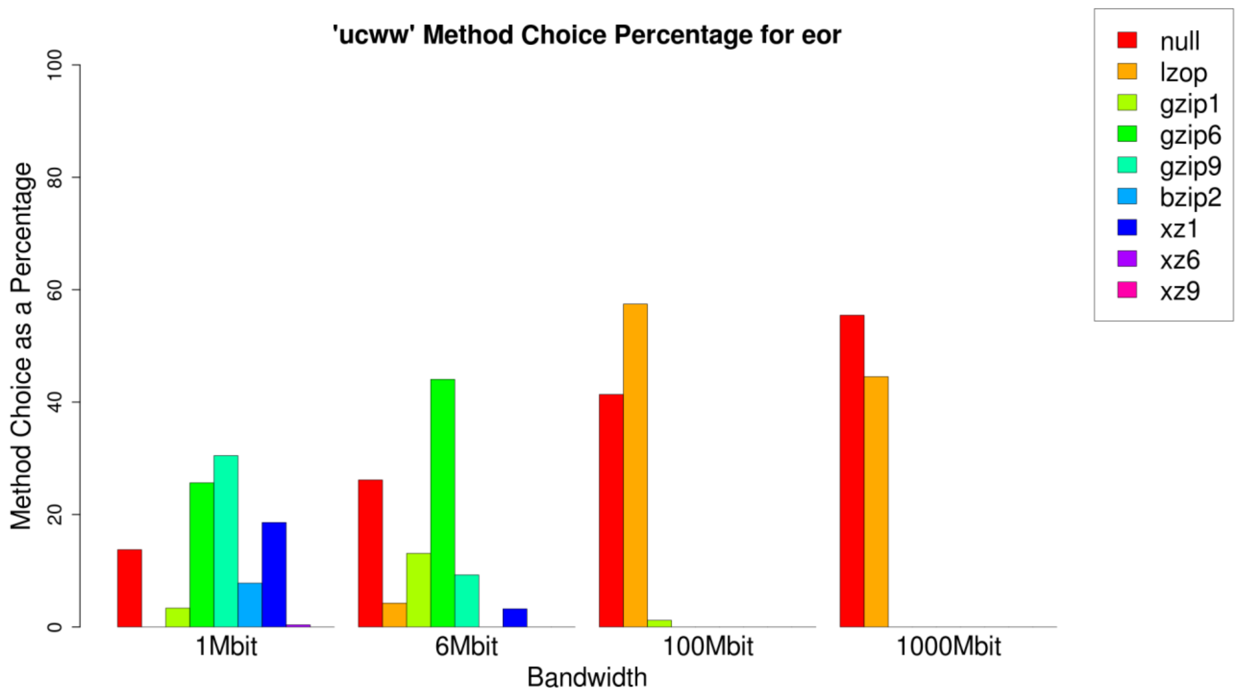


Figure 45. Method choice for User Web data at multiple bandwidths.

9.4.5 User File Data

In the same way that User Web Data is similar to Facebook data in terms of results, User File Data results (shown in Figure 46 through Figure 49) are similar to YouTube data, with the exception that User File data is slightly more compressible, particularly with strong Methods. Unfortunately, strong Methods are not helpful at higher bandwidths.

We can surmise this from the throughput data because "best EOR" performs slightly better for User Files than YouTube at low bandwidths and because the Method choice counts show strong compressors well represented at 1 and 6Mbit/s but absent at higher levels. Finally, like YouTube, User File data shows a clear preference for large chunk sizes at higher bandwidths. Comparing with the dzip results (Section 8.4) shows that YouTube data compresses to around 97% with gzip-4 through -9, while User File data best compresses to ~93% using xz-4 through -9.

During the dzip evaluation I did not perform tests to evaluate the ability of stronger compressors (such as bzip2 or xz) to compress User Files with chunk sizes larger than 512K. However, a similar test using xz-9 on the 50 megabytes of concatenated User File data shows that it can be compressed down to 70.6% with sufficient amounts of input to compress. This suggests that it is worthwhile to investigate using larger chunk sizes not just to improve CR in space-prioritizing tasks, but also to potentially improve throughput of data similar to the User File experimental data.

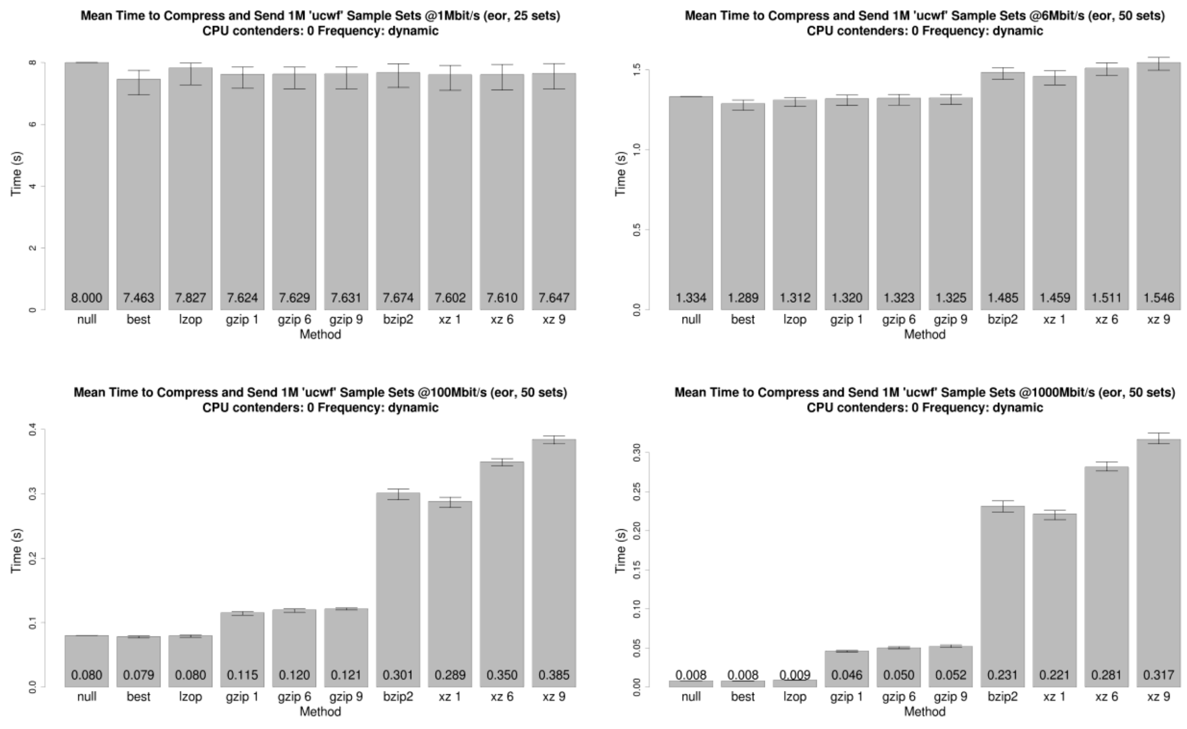


Figure 46. Mean time to send 1M User File data at various bandwidths.

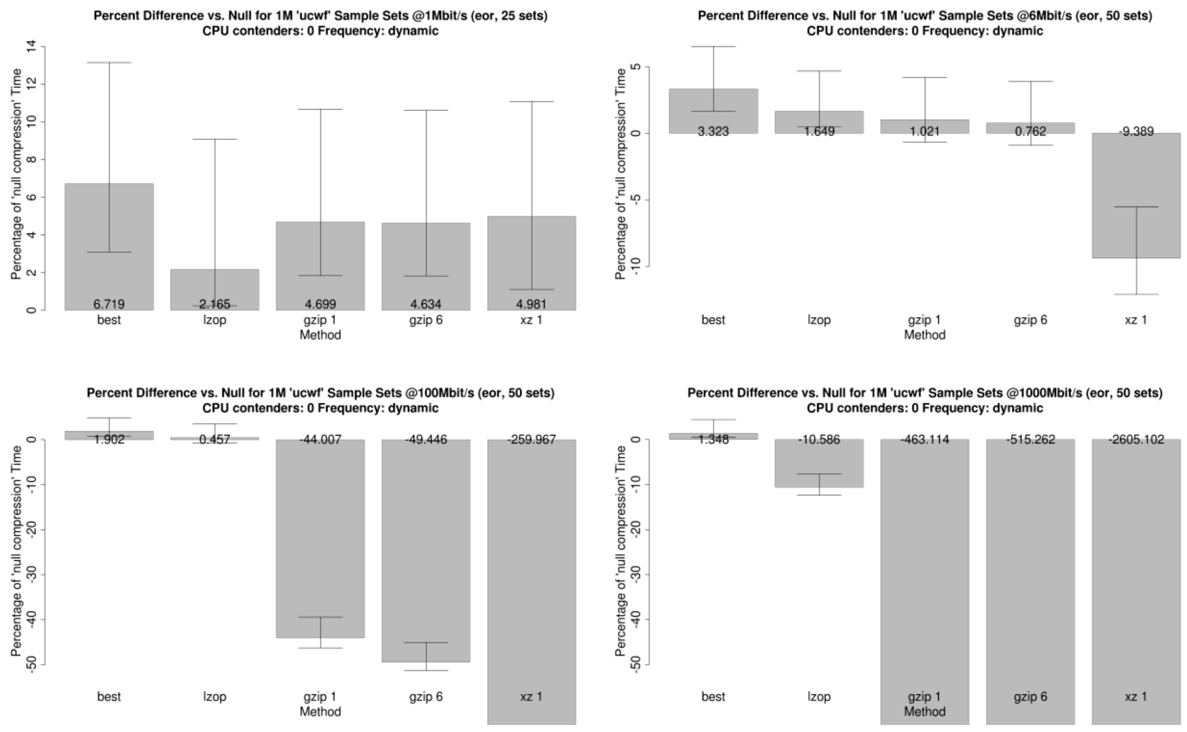


Figure 47. Percent difference in mean time by bandwidth for User File data.

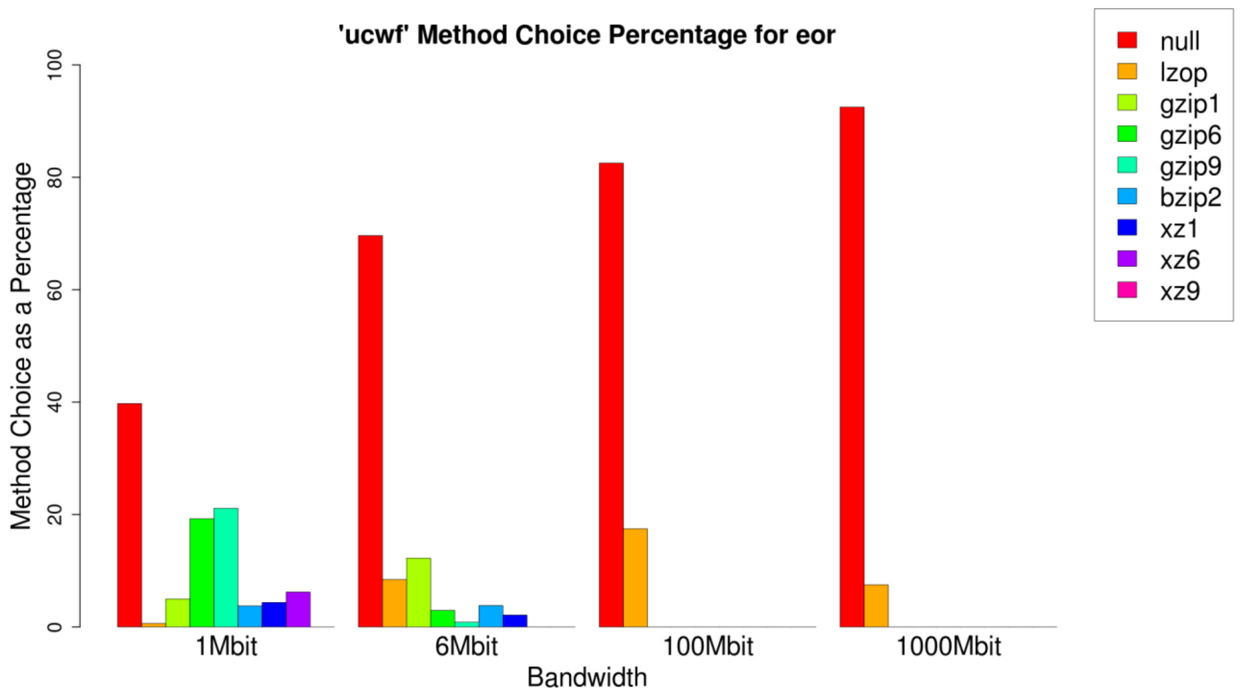


Figure 48. Method choice for User File data at multiple bandwidths.

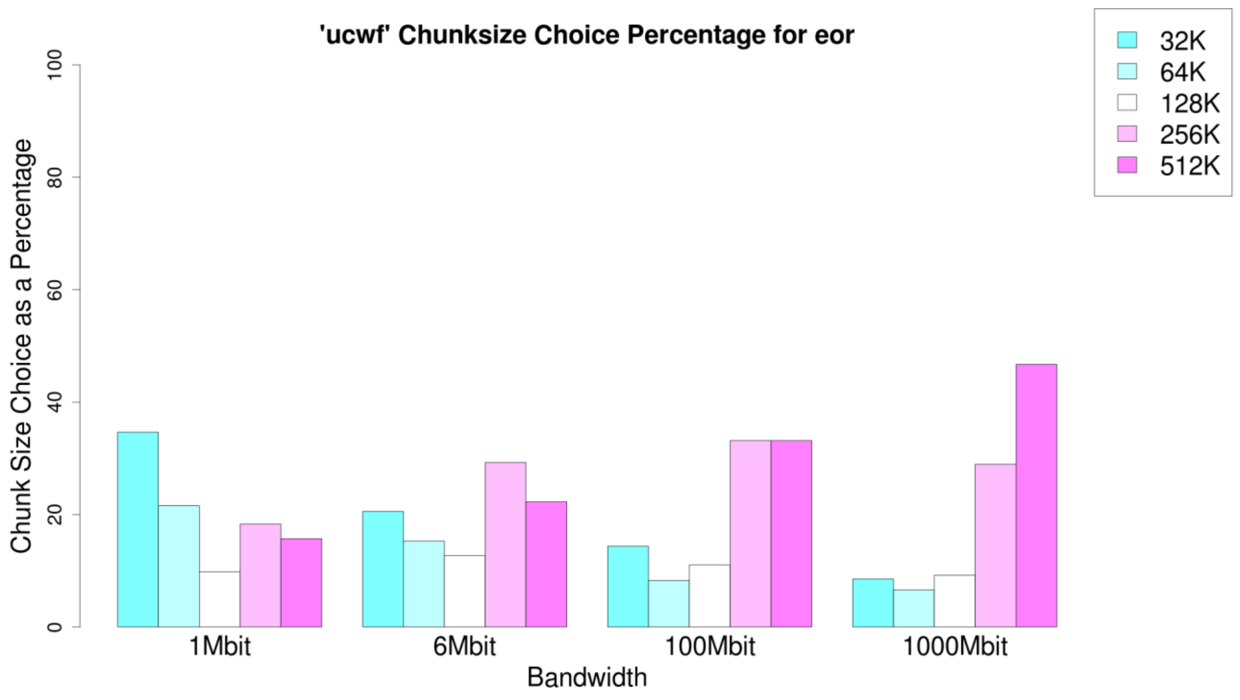


Figure 49. Quantization level choice for User File data at multiple bandwidths.

9.4.6 Binary

Of the data types explored in this project, binary data (data sampled from system binaries) is most like the Wikipedia data in terms of its compressibility and performance in conjunction with compression Methods. It is obviously highly compressible, as is evidenced by the large difference between the time for null and that of "best," which at low bandwidths saves more than half the time of null compression. While the differences between mean runtimes are not always significant (especially for lower bandwidths where strong compressors have more time to compete), the time results suggest that the best strategy for binaries is different at each bandwidth level.

Figure 50 shows the mean times for our four main bandwidths and set of Methods. At 1Mbit/s, "best" and xz-1 have almost identical mean times, which suggests that xz-1 is the best strategy at this bandwidth. At 6Mbit/s, the "best" mean runtime is somewhat less than all other Methods (although the differences are not statistically significant). This suggests that the best strategy is made of a combination of Methods. At 100Mbit/s, best's runtime is virtually identical to lzop's, suggesting that lzop dominates the best strategy at this bandwidth. Finally, at 1Gbit/s, the best strategy is again lower than any other Method, and this time the difference is statistically significant.

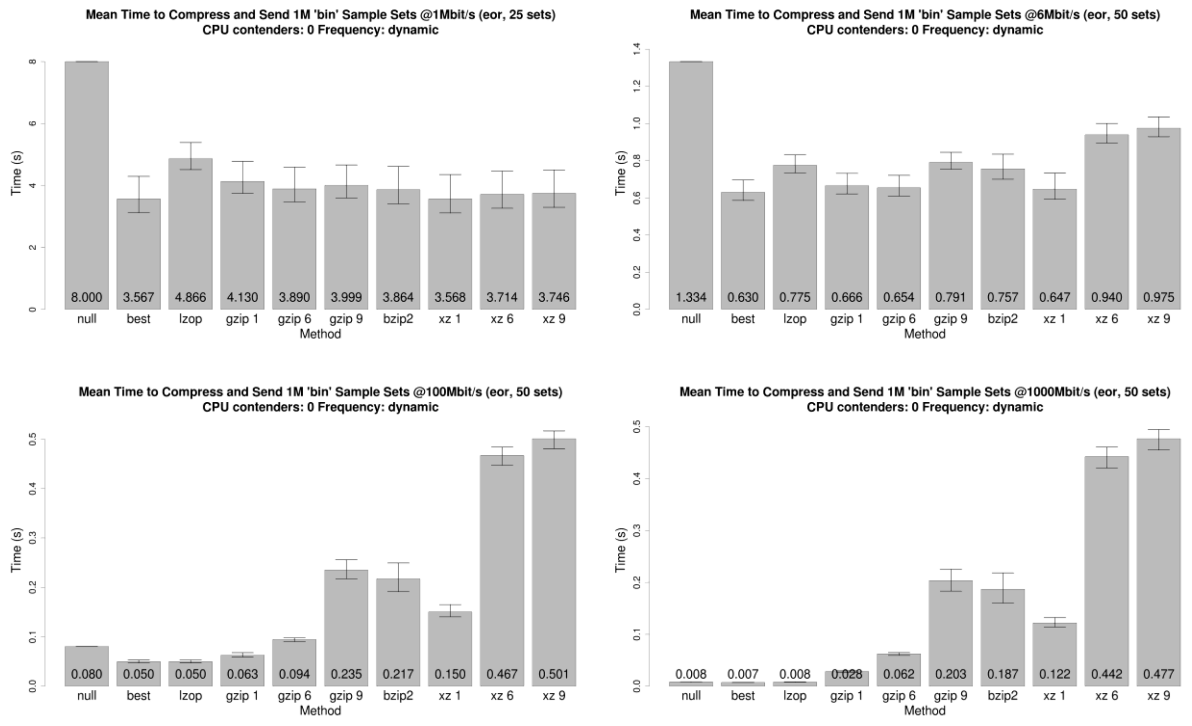


Figure 50. Mean time to send 1M Binary data at various bandwidths.

Before we examine the Method choices for this data, it is worthwhile to "zoom in" at the percentage differences between the main set of Methods and null compression, shown in Figure 51. These results suggest that, while compressible, binary data is not as compressible as Wikipedia. At every bandwidth, the benefit of compression for Wikipedia is larger than for binary data. Additionally, fewer Methods manage to be profitable at higher bandwidths for binary data. For example, gzip-6 is unable to be profitable at 100Mbit/s (or above) on binary data, and lzop is unable to be profitable at 1Gbit/s. Furthermore, the penalties faced by failing Methods are comparatively larger for binary data. Thus, we can conclude that compressing binary data is a "harder case" than Wikipedia.

If we were seeking to create a spectrum of data types in terms of their amenability to compression for throughput, we might intuitively place binaries between Wikipedia and Facebook data. Binaries were compressible down to 37% in the dzip compression ratio results (Figure 17), while Facebook data was only compressible down to ~62% (Figure 20). However, this kind of strict performance hierarchy is not supported by the data; in some ways binaries are easier than Facebook data, in some ways harder, and when and why this is the case depends on multiple opportunity factors. Comparing percent improvement for binaries with Facebook (Figure 36), we see that binaries are more "improvable" than Facebook, up to 100Mbit/s. However, at 1Gbit/s, Facebook data (best = +14.3%) is more efficient than binary data (best = +11.3%). Additionally, while lzop is still successful on Facebook at 1Gbit/s, it is not successful for binaries. Looking at Method choices can help explain why.

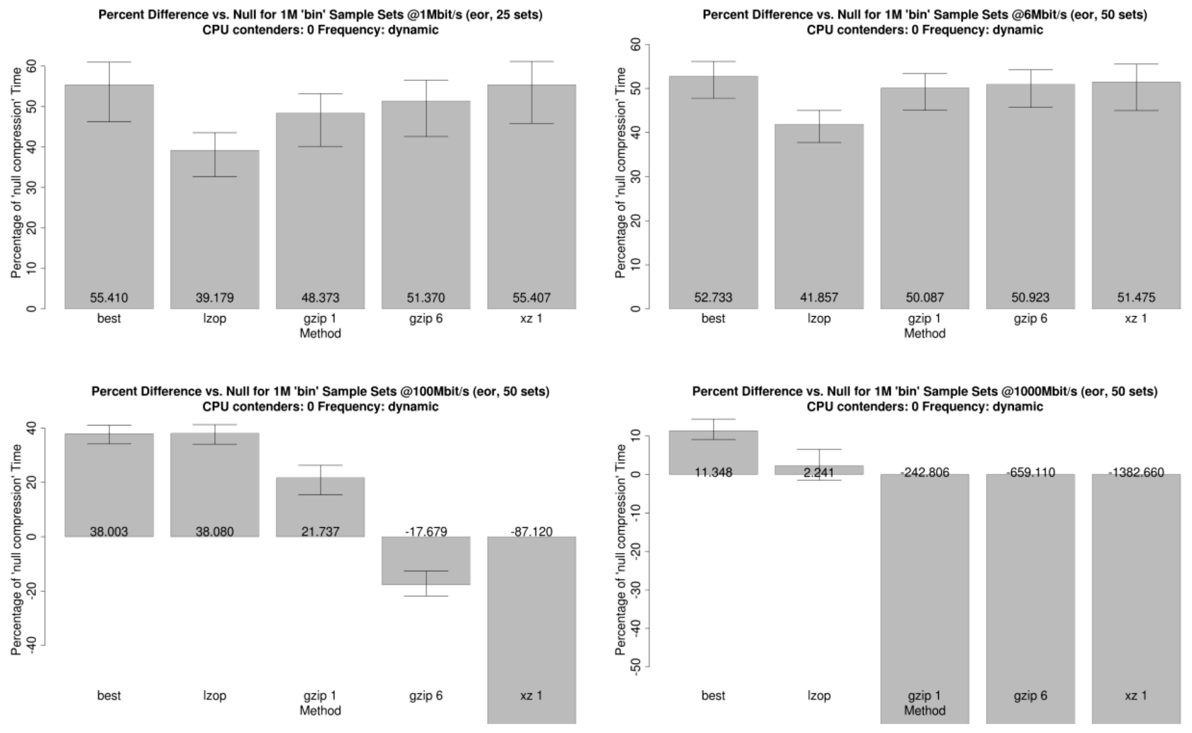


Figure 51. Percent difference in mean time by bandwidth for Binary data.

It appears that the "best" strategy for binaries is indeed different at every level. Figure 52 shows that at 1Mbit/s, the most used strategy is indeed xz-1, while at 6Mbit/s it is a combination of xz-1 and the faster gzip-6. At 100Mbit/s, gzip is mostly too slow to be the best choice, and lzop takes its place as the only alternative. However, the dzip compression ratio results for binaries (Figure 17) shows that lzop is *statistically worse* than all the other Methods at compressing binary data. In an experimental environment where most alternatives are indistinguishable, this is a big fault.

At 1Gbit/s, lzop's poor compression has become enough of a burden that the proportion of null compression operations quadruples. This *further* hurts the performance at 1Gbit/s (because the data is now virtually half uncompressed). In other words, while switching to lzop from some other compressor due to increasing bandwidth allows compression to continue to be a win, binaries still end up performing worse than Facebook because lzop is not very effective for them. In the end, the ostensibly more-compressible binaries are actually less amenable to compression at high throughputs.

Finally, the fact that xz puts in such a strong showing for low bandwidths suggests that binaries could also benefit from being compressed in larger chunk sizes. A simple test on the 50-megabyte concatenation of binary test samples results in an astounding CR of 0.18 for binaries with xz-9. Of course, xz-9 is so slow as to not be competitive at higher bandwidths. However, if an effective chunk size was combined with parallelism, it is possible that xz could be made effective at higher bandwidths (if not the highest bandwidths).[88]

[88] Chunk size choices for binary data are omitted here because they are unremarkable.

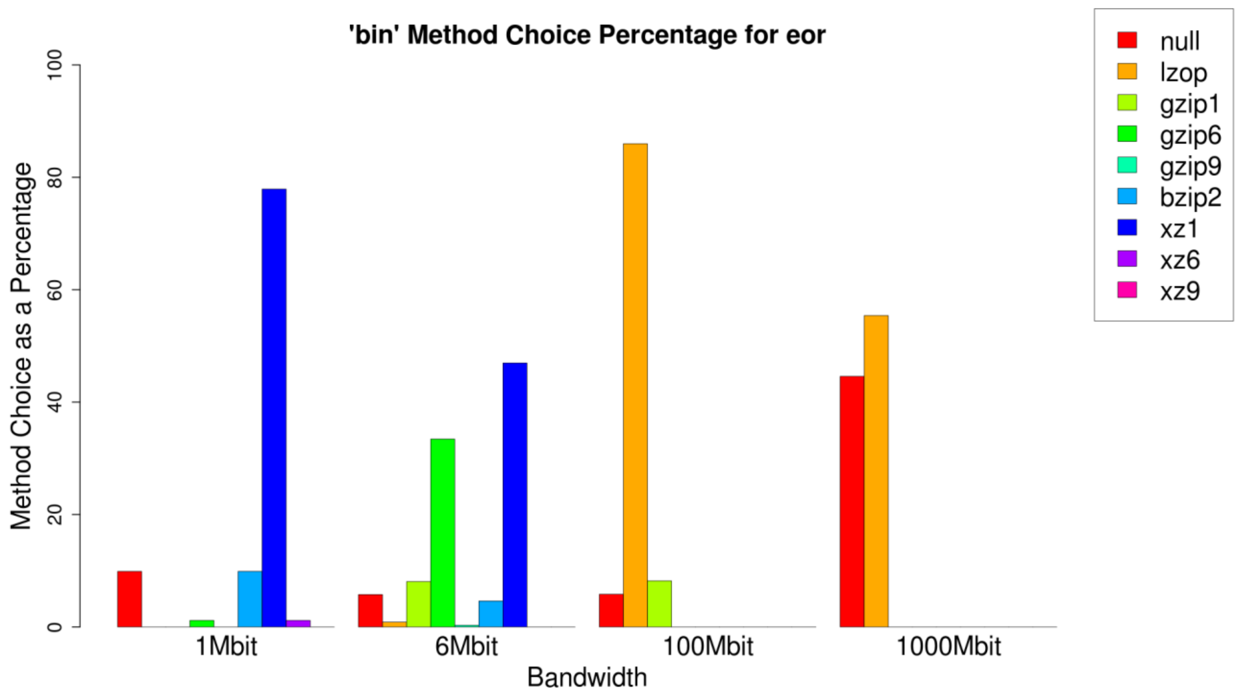


Figure 52. Method choice for binary data at multiple bandwidths.

9.4.1 Mail

Mail is another type of data with unique compressibility characteristics, which result in what may seem like counter-intuitive results. At first thought, email seems like it would be easy to compress, because it is mostly text. However, byte for byte, most stored email bytes are actually attachments. This means that a stream of "email bytes" consists of attachments, which are mostly not compressible, interspersed with smaller sections of text, which are compressible. Furthermore, mail is a data type that is almost completely uncompressible by lzop, as shown in Figure 18. Because lzop is the only compressor able to succeed meaningfully at gigabit speeds, one should immediately suspect (having read the previous information in this work) that email would be difficult to compress effectively at that rate.

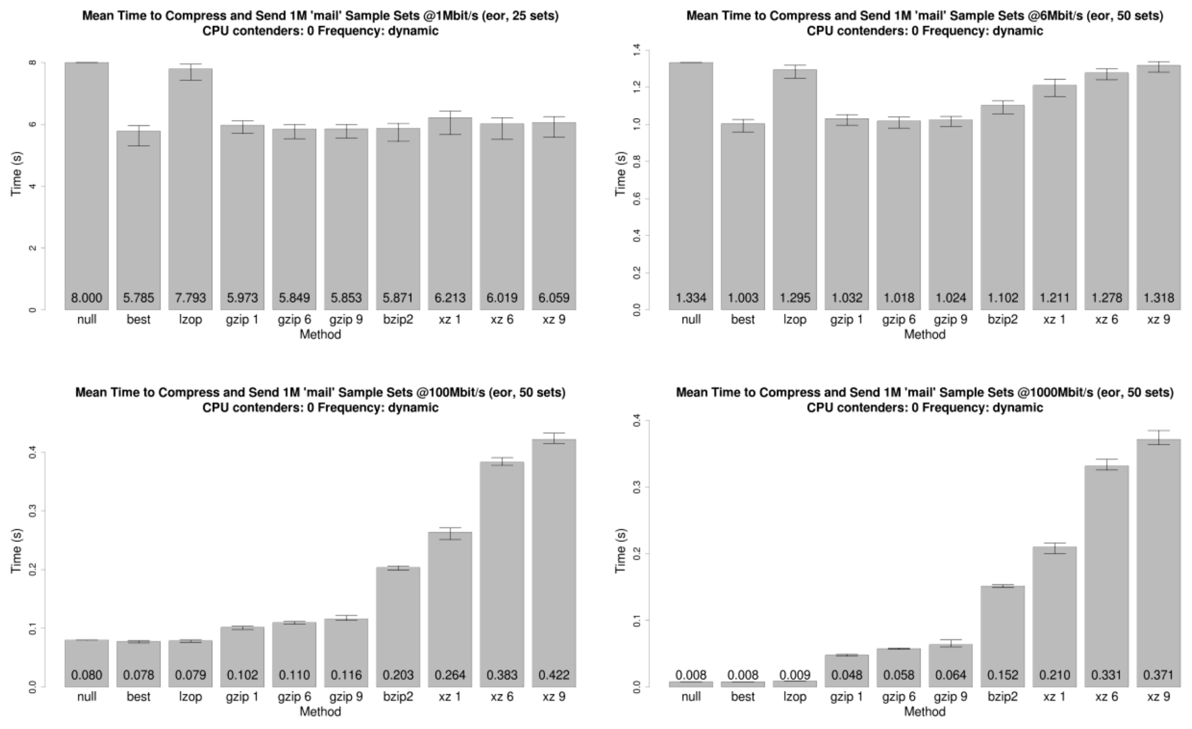


Figure 53. Mean time to send 1M User Web data at various bandwidths.

The data in Figure 53 confirms those suspicions. At 1Mbit/s and 6Mbit/s, lzop is barely an improvement over no compression. Best achieves a meaningful improvement over null, and at these rates the mean values are slightly better than any single Method (although the difference is not statistically significant). Past experience suggests that this is an indicator of a "hybrid" Method – some combination of one or more Models – being "best." At higher bandwidths, the stronger compressors are not able to keep up with the data, and because lzop compresses mail so poorly, "best" is *virtually* equivalent to null compression and lzop – unlike other compressible data types, there is not much improvement.

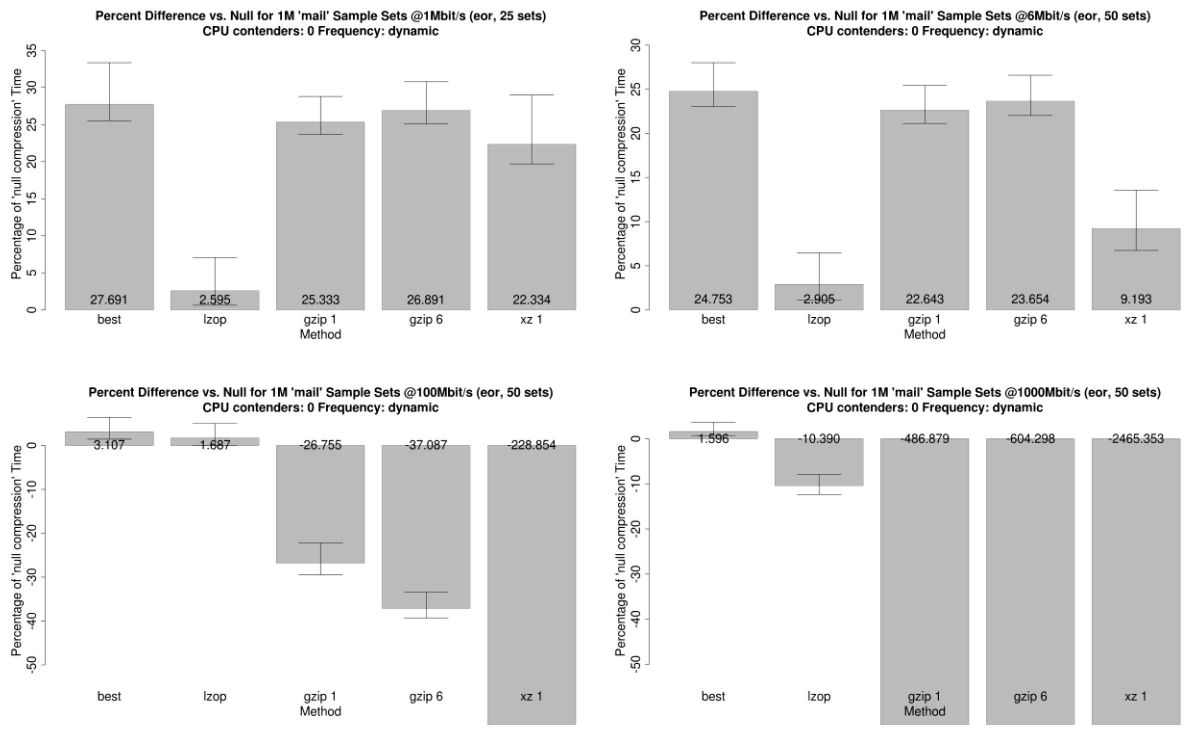


Figure 54. Percent difference in mean time by bandwidth for mail data.

I said "virtually" equivalent for a reason. Figure 54 shows us the detail of our main Methods as difference percentages. At the low bandwidths, Comptool is able to improve throughput by between about 25-28%, although this difference is not significantly different than gzip-6. At higher bandwidths, "best" manages to eke out small improvements -- ~3% for 100Mbit/s and 1.6% at 1Gbit/s – while lzop and the other compressors are either indistinguishable from null compression (e.g., lzop at 100Mbit/s) or are significantly worse than null compression (everything else).

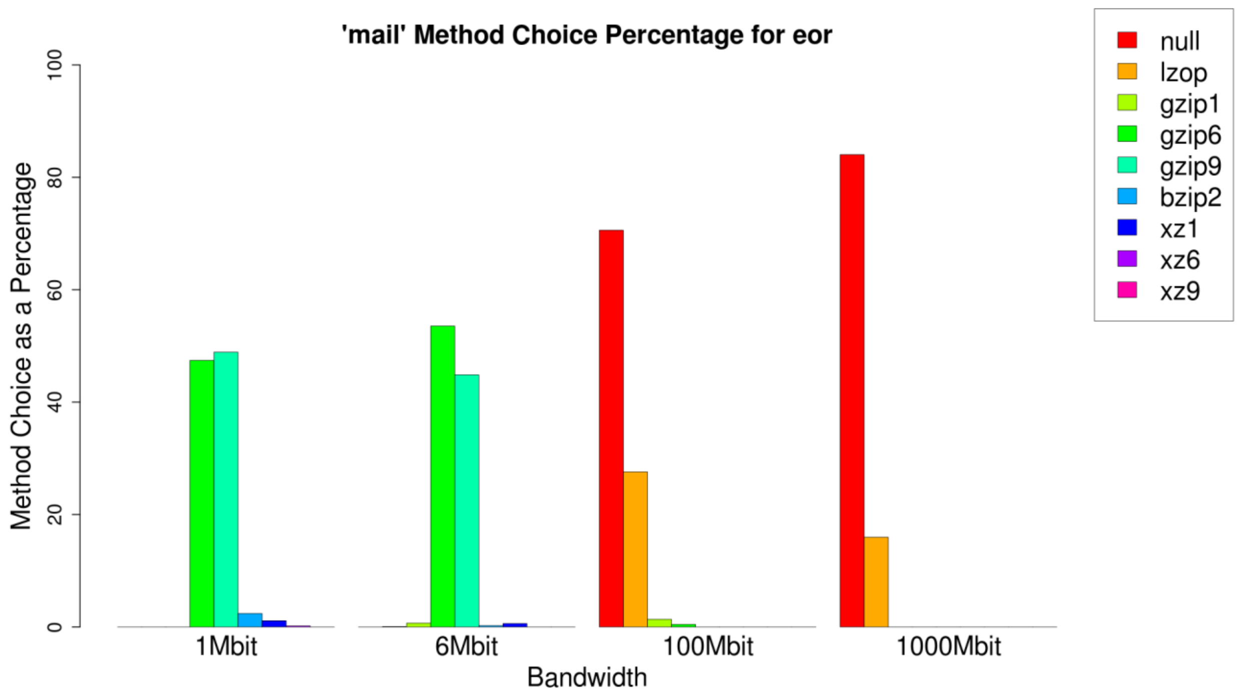


Figure 55. Method choice for mail data at multiple bandwidths.

Examining the Method choice data for mail (Figure 55) confirms our suspicions once again. The best strategies at 1Mbit/s and 6Mbit/s are a combination approach based primarily on gzip-6 and gzip-9. At 100Mbit/s and higher, lzop is the only compressor able to meaningfully contribute. Additionally, since lzop's CR is so poor for this type of data, both high-bandwidth strategies are composed primarily of null compression.

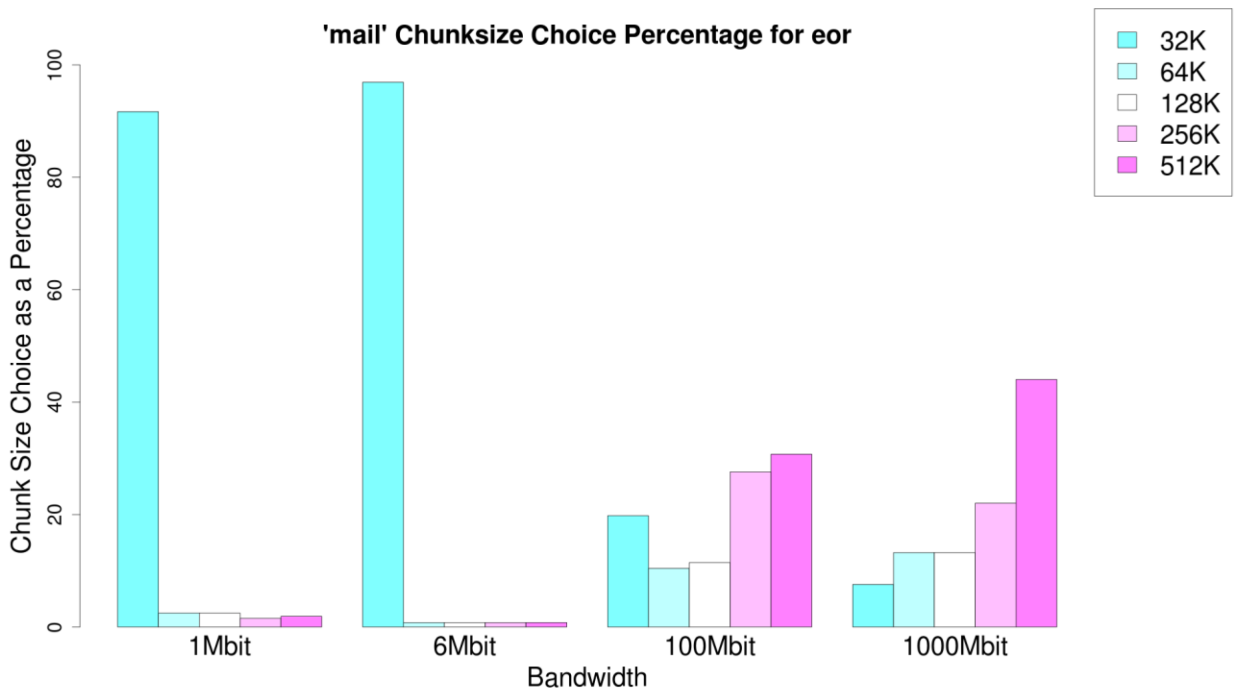


Figure 56. Quantization level choice for mail data at multiple bandwidths.

The quantization levels for mail, shown in Figure 56, tell an interesting story. The Method count plot showed that at the first two bandwidths, the best strategy for mail is composed almost entirely out of zlib operations. Internally, zlib compresses data in blocks of 32K; as a result, compressing larger sizes provides very little benefit in terms of compression ratio. Since at 1Mbit/s and 6Mbit/s there is surplus time for compression operations, using larger chunk sizes with gzip would only result in “trampling” better opportunities for compression with other Methods. Thus, the greedy “best” heuristic ends up overwhelmingly choosing 32K input chunks for bandwidths where gzip is dominant. On the other hand, as all compression Methods fail, there is little benefit to using smaller chunk sizes, so 512K comes to dominate the chunk sizes at 1Gbit/s.

Finally, like binaries, email is significantly compressible by xz-9 with large inputs. In my test compression of the sample sets concatenated together, xz-9 compressed 50 megabytes down to 16.8 megabytes, achieving a CR of 0.337, significantly better than the CR of 0.715 that xz-9 achieved on 1M samples for the dzip evaluation (Figure 18). Accordingly, it would be worthwhile to explore using large input sizes with these Methods in addition to parallelism to improve throughput.

9.4.2 Zero

"Zero data" – files constructed of nothing but binary zeros serve as our best-case scenario input. Because the entire input consists of the same byte, it should compress very well with every Method. Most Methods are also able to compress best-case input like this very quickly. Figure 57 shows the mean times to compress and send 1M samples of zero data at our bandwidths.

Obviously every compressor is able to significantly improve performance at 1Mbit/s and 6Mbit/s. However, at 100Mbit/s, xz-6 and xz-9 are unable to keep up. At 1Gbit/s, bzip2 and all xz Methods are slower than null compression. Best is indeed the best, although the differences are difficult to depict in a graph. For 1Mbit/s and 1Gbit/s, "best" is just slightly faster than lzop; for 6Mbit/s and 100Mbit/s they are statistically equivalent.

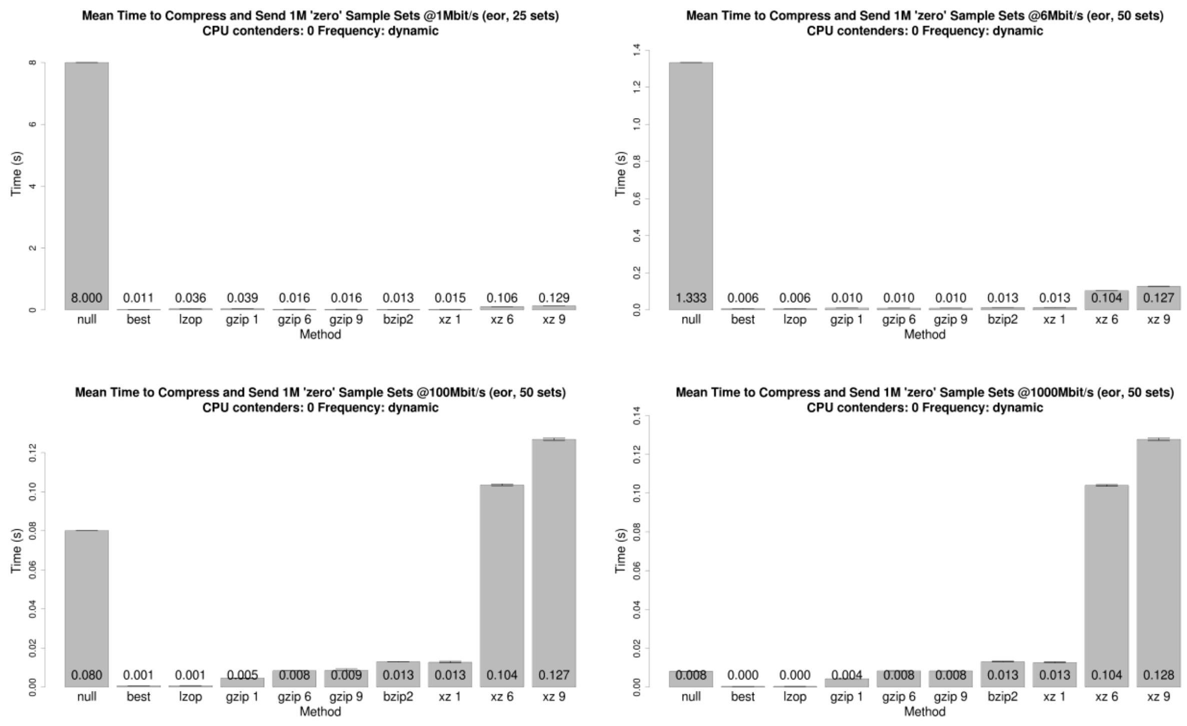


Figure 57. Mean time to send 1M zero data at various bandwidths.

Figure 58 shows the percentage differences, but due to the simplicity of this scenario, it is difficult to see any difference between the Methods at 1Mbit/s and 6Mbit/s. However, at 100Mbit/s the degrading performance of gzip and xz are evident. At 1Gbit/s, “best” and lzop are statistically tied for the best improvement. Gzip-1 is able to remain profitable even though both gzip-6 and xz-1 are now degrading performance.

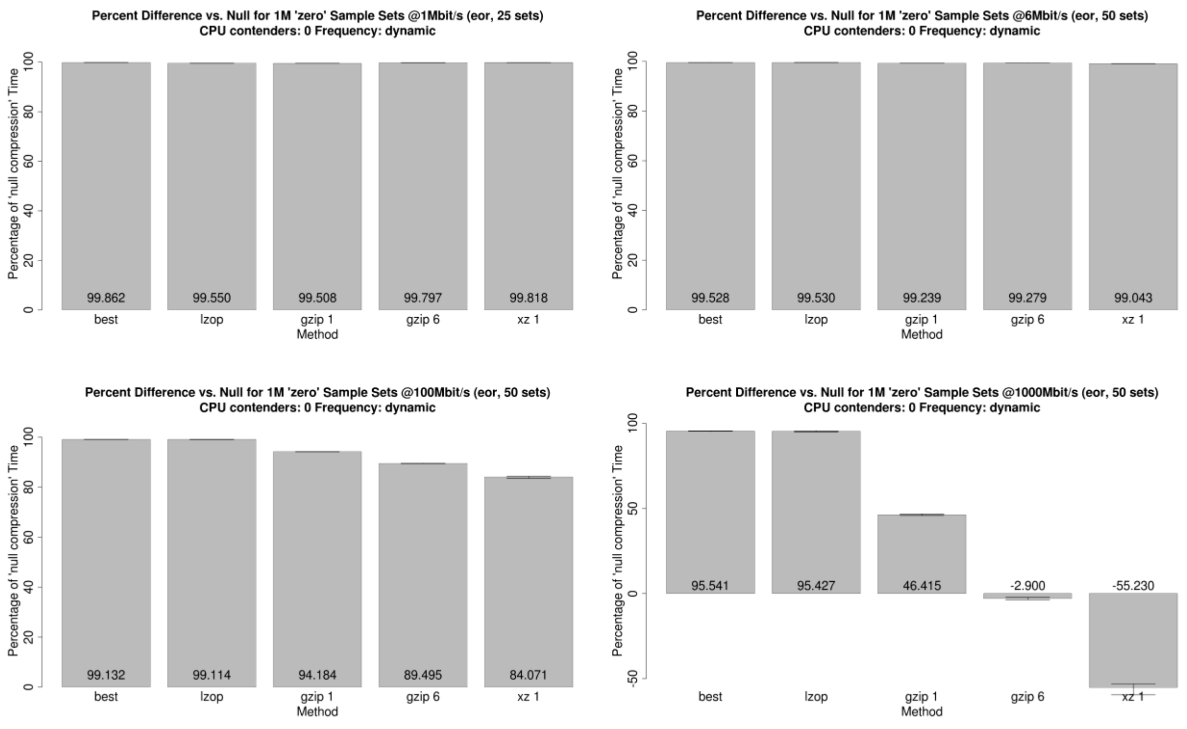


Figure 58. Percent difference in mean time by bandwidth for zero data.

Even though the Method counts for zero data are extremely simple, they provide interesting information about choices for AC. At 1Mbit/s, bzip2 is the best algorithm because it has the best CR for zero data (as described in Section 8.4). By 6Mbit/s, bzip2's AOR clearly cannot keep up with the bandwidth, even though the workload is so simple. (And the CRs for all algorithms are so similar that bzip2's having the best CR is not that valuable.) However, the second-best Method in terms of CR was xz-1 (also described in Section 8.4). That xz-1 is *not* the Method selected for use at 6Mbit/s is indicative of its large time cost (visible in Figure 57 as a runtime more than twice as slow as "best"). Instead, lzop is the choice made for zero data at the remaining bandwidths.

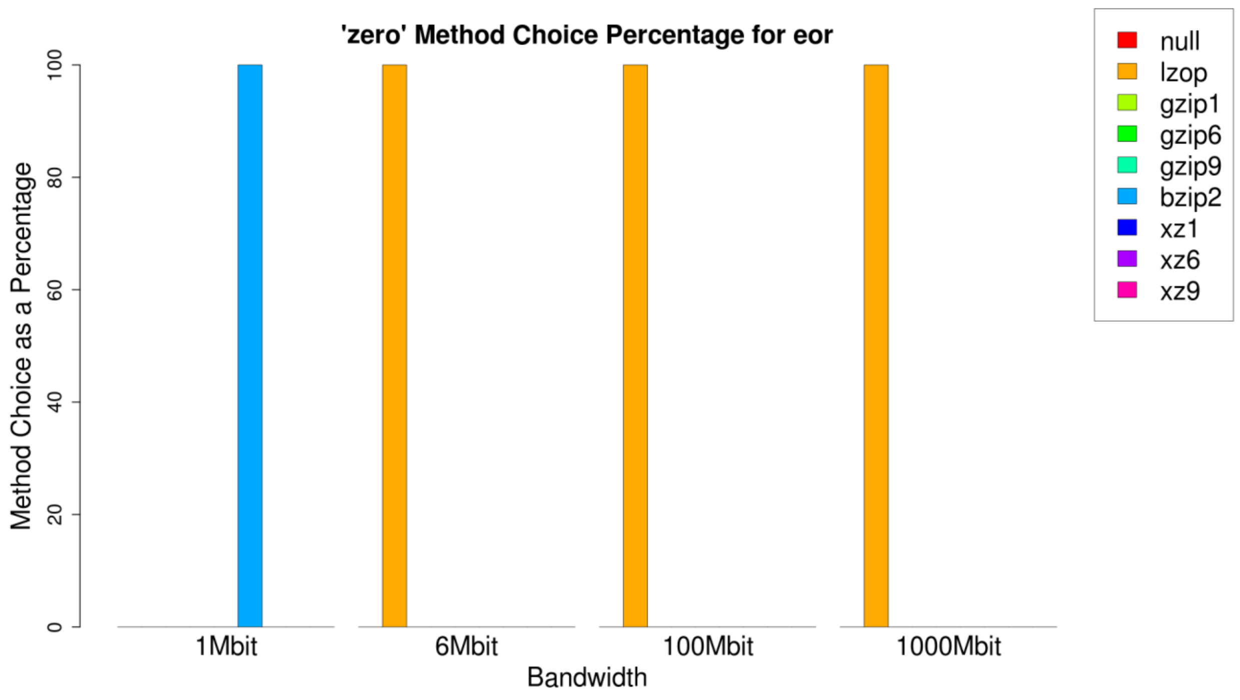


Figure 59. Method choice for zero data at multiple bandwidths.

Chunk size selections for zero data are shown in Figure 60. They are interesting primarily because the single-Method nature of the Method choices just described allows us to see the "preferred chunk sizes" of the single Methods in this scenario. As Figure 60 shows, bzip2 (chosen at 1Mbit/s) chose almost exclusively 256K chunks. Lzop instead overwhelmingly chose 512K chunks, although in a smattering of events it chose smaller sizes (not including 32K). This helps to illustrate that no single chunk size is ideal for compression across all Methods.

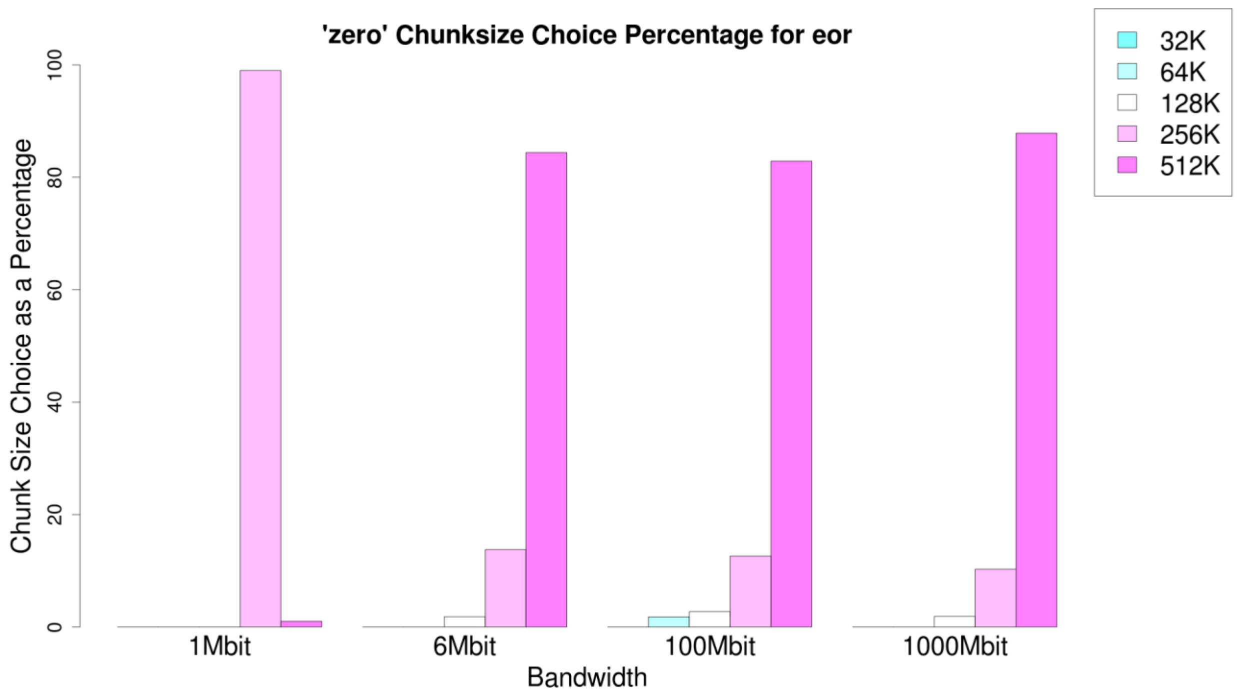


Figure 60. Quantization level choice for zero data at multiple bandwidths.

9.4.3 Pseudo-random

Finally, pseudo-random data serves as our worst case scenario. Because it is not compressible, we would expect all Methods to be wasteful. Regardless of bandwidth, the best strategy should be null compression. Figure 61 suggests that this is the case; at each bandwidth, "best" has an identical runtime to null, and all other Methods take longer.

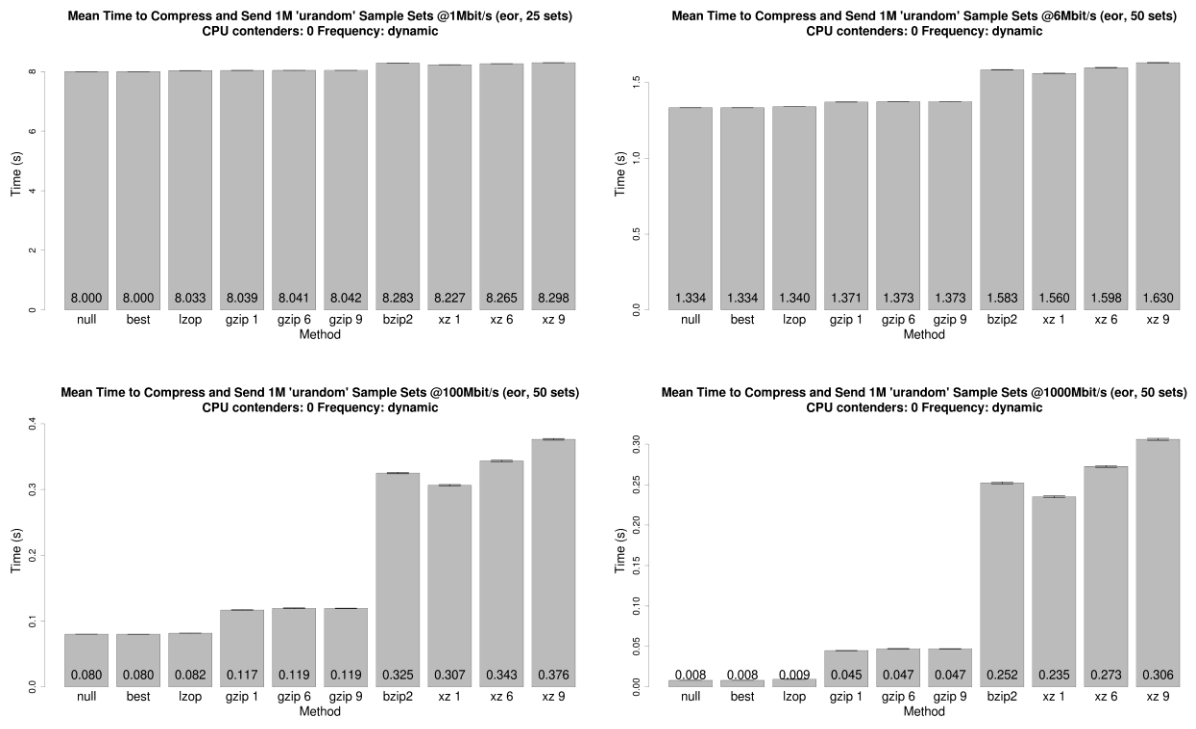


Figure 61. Mean time to send 1M pseudorandom data at various bandwidths.

Percent improvement differences in Figure 62 show greater detail. In particular, the overhead of the best approach is visible. At 1Mbit/s, Comptool reports no difference. At 6Mbit/s and 1Gbit/s, Comptool reports that "best" is 0.001% and 0.132% better than null, although the CIs for both these scenarios span zero. Interestingly, at 100Mbit/s, the 0.28% improvement of throughput achieved by "best" is actually statistically significant.

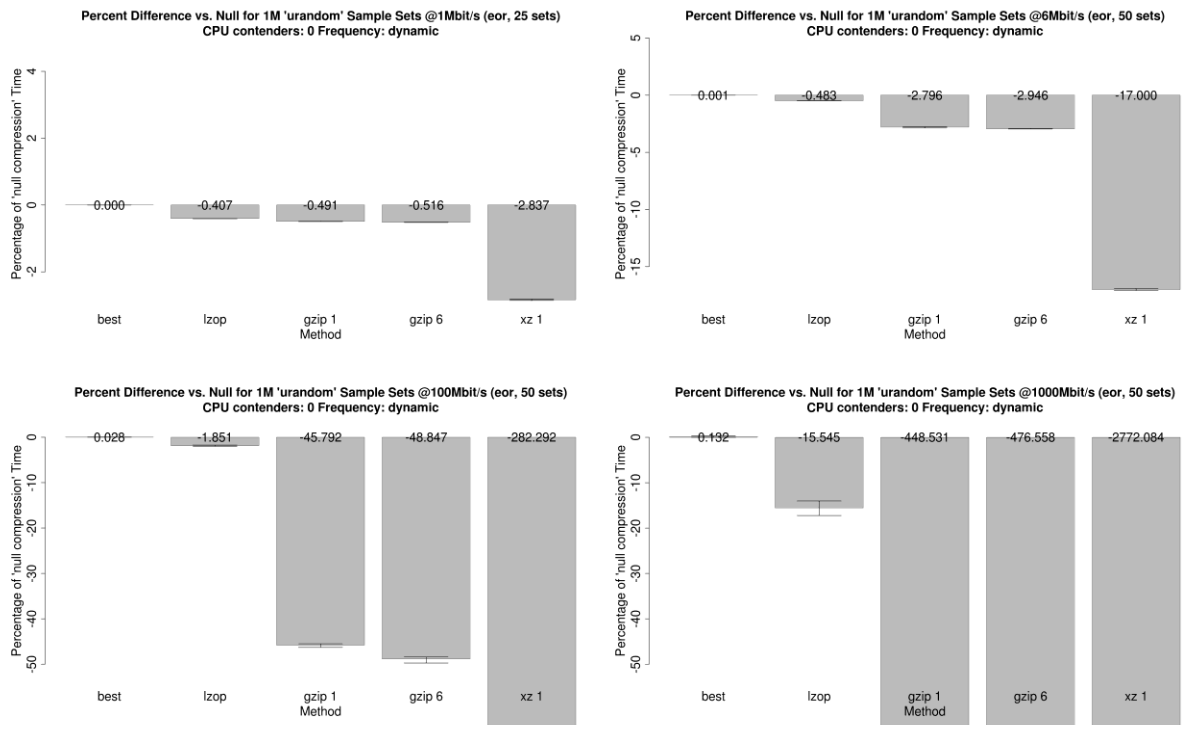


Figure 62. Percent difference in mean time by bandwidth for pseudo-random data.

While the 0.132% improvement at 100Mbit/s may be statistically significant at the 95% level, this is almost certainly an error belonging to that unlikely 5%. We can be fairly certain of this because, as expected, Comptool chooses only null compression at every bandwidth. The only other conceivable reason that a "best" strategy composed of nothing but null compression could beat a *static* (i.e., non-quantized) strategy of null compression is if some esoteric system effect enabled the discrete chunks of "best" to complete slightly (i.e., 0.132%) faster than a "bulk" write. Further research into these issues could enable the identification and tuning of parameters like this that could impact performance.

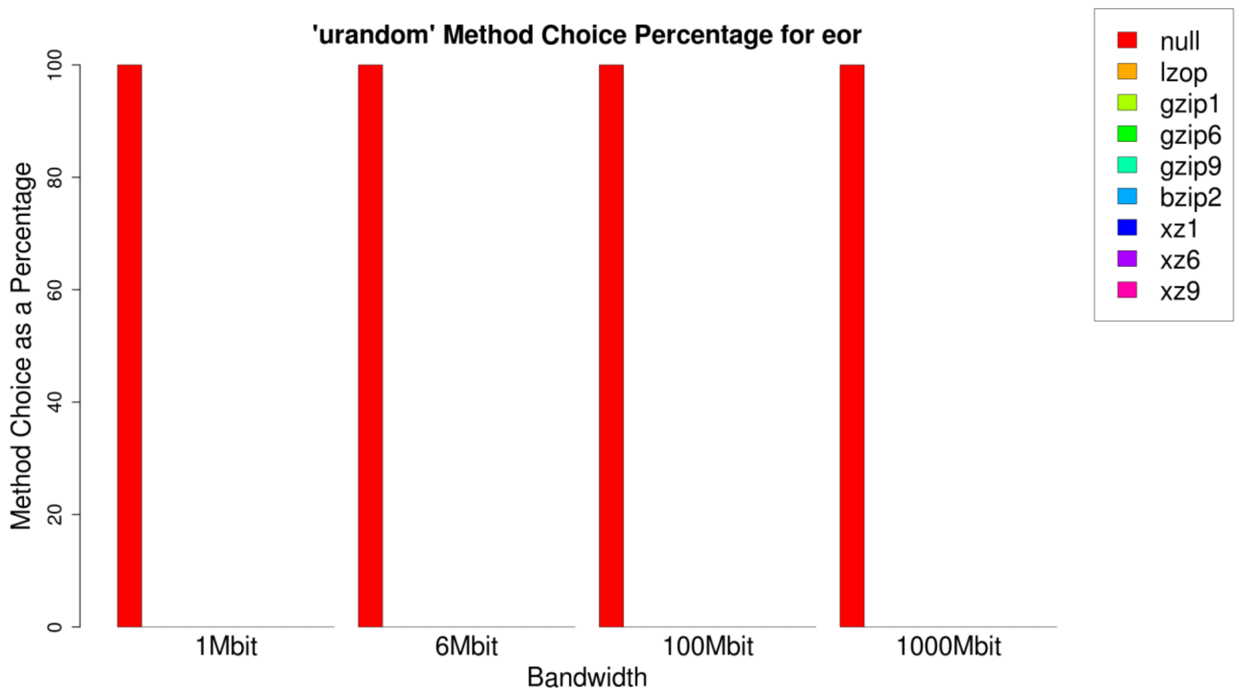


Figure 63. Method choice for pseudo-random data at multiple bandwidths.

Surprisingly, the chunk size choices for pseudo-random data (shown in Figure 64) are actually interesting. While it is not surprising that null compression is universally selected, it is surprising that a variety of chunk sizes are chosen and that the maximum chunk size is not the most common choice. Additionally, the selection of choices does not appear random, and they are generally similar regardless of bandwidth. Put together, this information suggests that the choice of these chunk sizes (especially the dominance of 256K chunks) are due to some system effect, such as kernel buffering, Python internals, or some other esoteric reason. Regardless of the reason, it demonstrates that chunk sizes are not always selected to conform to human intuition and it suggests that obscure lower-level system effects (which may be based in hardware or software) can affect coarse Method choices in surprising ways.

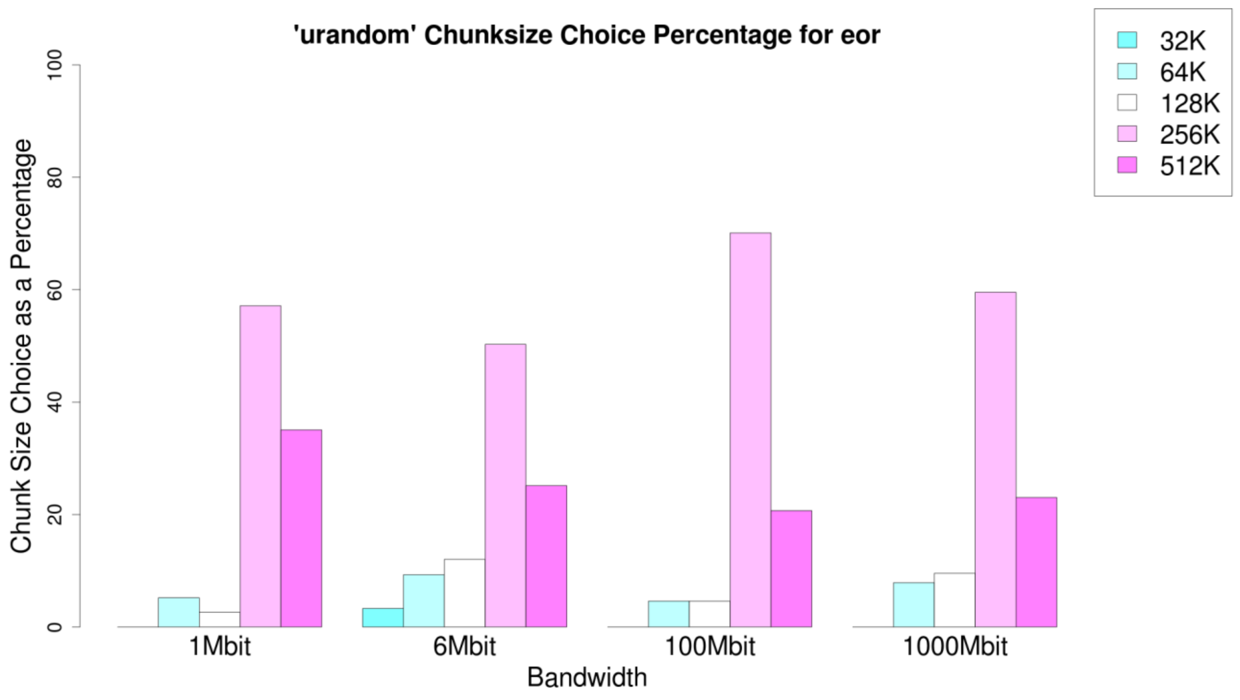


Figure 64. Quantization level choice for pseudo-random data at multiple bandwidths.

9.5 CPU Frequency and Contention

CPU frequency and CPU contention had visible effects in my experiments, primarily in terms of mean runtimes. As one would expect, an increase in CPU frequency led to improved performance for certain event types. Conversely, an increase in CPU contention (i.e., load) led to a decrease in performance for certain event types. I chose not to include these plots because in my experiments, the effects of these factors were much smaller than those caused by the bandwidth, data type, Method, and chunk size. Nevertheless, even if they did not radically alter the best strategy for a given scenario, they were statistically significant factors in many cases. Additionally, the impact of factors like load and frequency are sure to be dependent on the underlying hardware (e.g., number of cores) and software environment (e.g., OS, nature of contending workloads). Thus, it is instructive to see when and how these factors affect outcomes.

Before discussing specific examples from the results, it is instructive to revisit our EEOR equation and our "rules of thumb," described in Section 9.3. EEOR depends on the current available bandwidth during the operation (ABW), the compressed output rate of the Method (AOR) for the given data and environment, and the compression ratio (CR) achieved by the Method on the data (in the form of the compression gain, CG). If the AOR for a given event is greater than the ABW, then the maximum throughput for the Method is the ABW multiplied by the CG. This is our "rule of thumb" #1. The second rule of thumb is that if the AOR is less than the ABW, then the AOR and the CG multiplied together will determine the output rate. In other words, given a set of Methods for which the AORs are all faster than the ABW, the best Method will be the one with the lowest CR (i.e., highest CG). From this perspective, Methods with AORs *less* than the ABW effectively suffer a penalty to their CG because the AOR disadvantage must be made up by the CG in order for the Method to be competitive.

We can assume that the local CPU frequency and system load do not, strictly speaking, affect the ABW of the channel[89] or the CR of the event. Thus, while small improvements are available across the board for reductions in load or increases in frequency, the only Methods that improve meaningfully with additional CPU cycles are those Methods that are computationally bound, that is, those Methods with AORs that are less than the ABW.

However, at least two factors keep these Methods from becoming especially successful when given a boost: First, the practical benefit of reducing load or increasing frequency is small relative to the AOR disadvantage that slow compressors face. In other words, the boost is not normally enough to make a difference. Second, that gap is difficult to bridge because the CRs for the Methods in these experiments are extremely similar. This means that even if the AORs

[89] Obviously there may be circumstances where changing hardware states or system load (e.g., kernel overhead) could affect throughput. However, for the purposes of this work I think this is a safe simplification to make.

were improved, the CRs available to those algorithms (at the input sizes used by Comptool) are not particularly impressive (see Section 8).

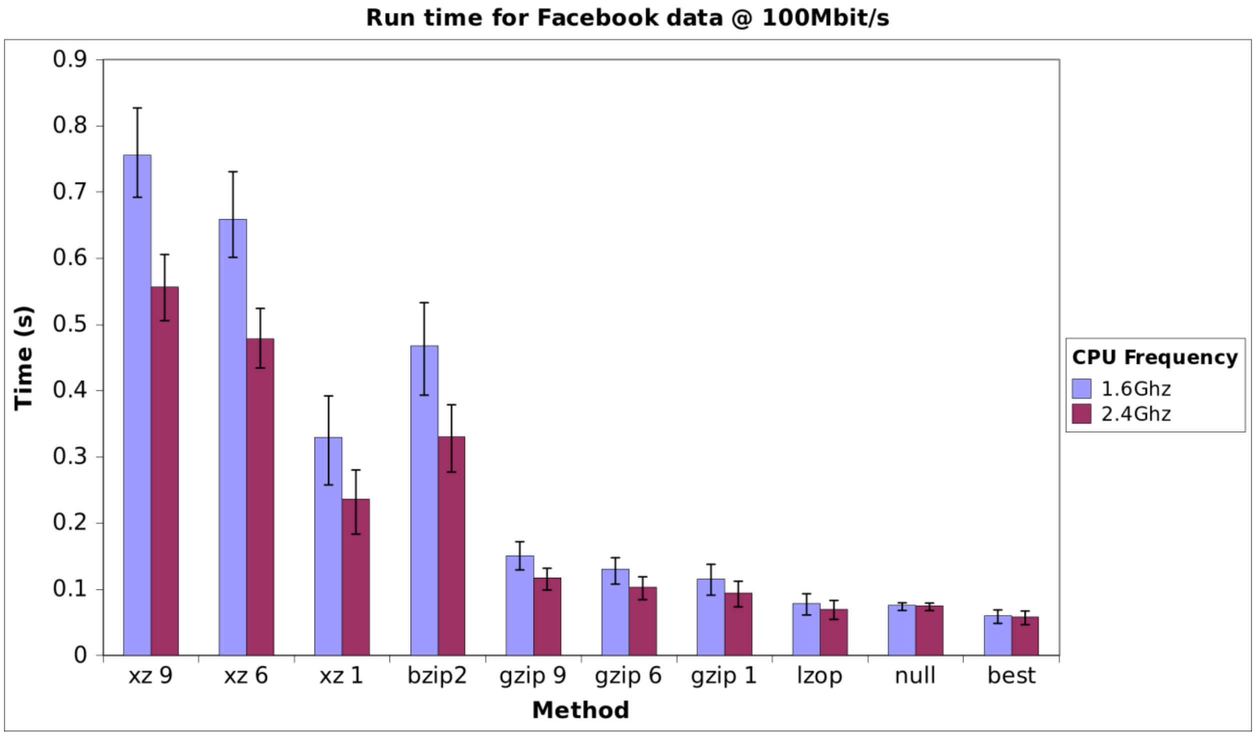


Figure 65. Example effect of CPU frequency.

Figure 65 shows a representative example of the effect of CPU frequency on a Comptool experiment using Facebook data at 100Mbit/s. There are significant improvements for "strong" Methods which were computationally bound. For example, xz-9, xz-6, and bzip2 all show significant improvements in time at the higher CPU frequency. However, the improvements diminish (and are less statistically significant) for Methods which were already strong performers. This means that even though some Methods were significantly improved, the performance of the "best" strategies are basically unchanged.

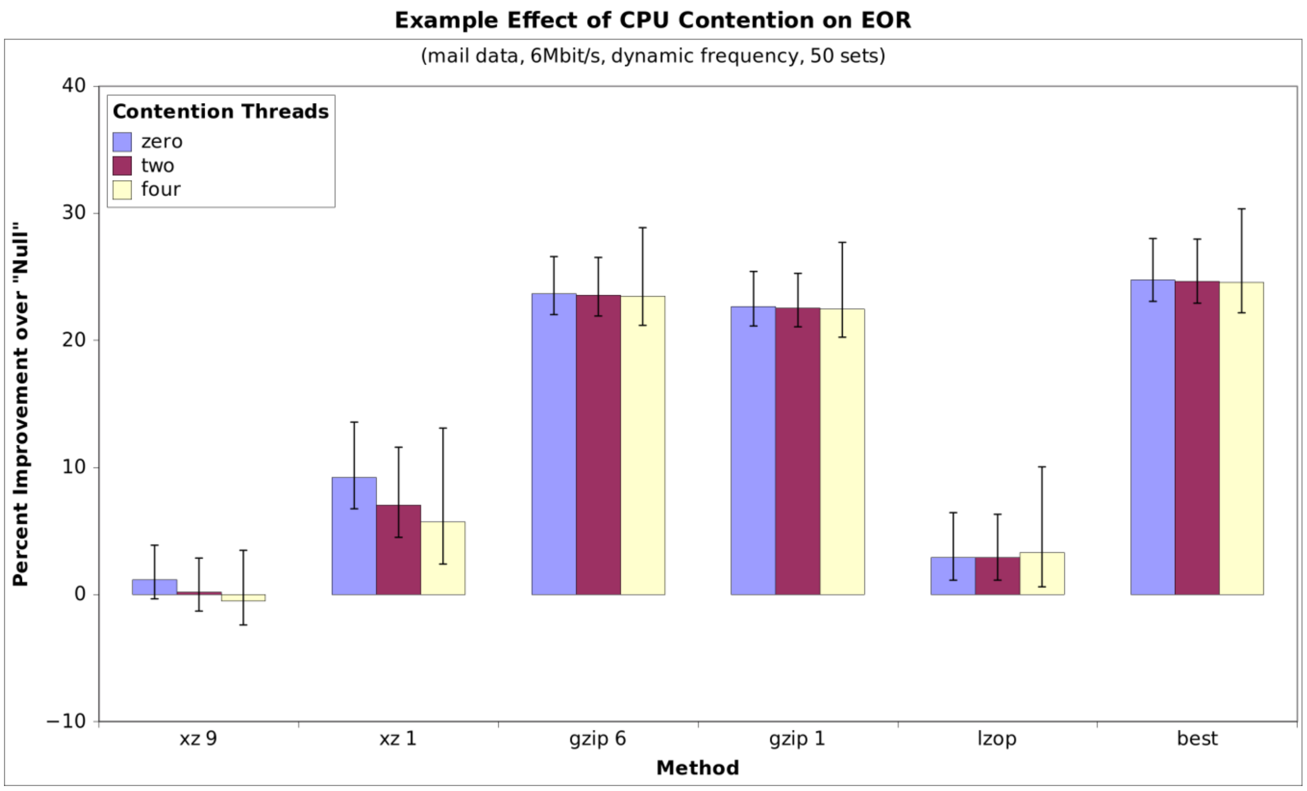


Figure 66. Example effect of CPU contention.

CPU contention also has an effect on results, although like frequency the effect is small and it is difficult to find an interesting example with statistical significance. Figure 66 shows mail data at 6Mbit/s with zero, two, and four CPU contenders. It appears as though the mean performance of xz-9 and xz-1 degrade as contention increases (and this result is representative of other experiments), but because the effect is smaller than the CIs, we can't say with confidence whether it is due to the increase in contention. This is exacerbated by the fact that CIs seem to *increase* with CPU load, which makes sense – a system's performance is bound to be more variable under load, which would increase the spread of individual measurements.

Because contention would result in a reduction of AOR (similar to a reduction in frequency), it is primarily an effect that impacts compression Methods which are already computationally bound or are on "the edge" of some performance boundary. I have only seen contention make a “significant difference” when a result was already extremely close to zero, and a change in contention affected whether the CIs spanned zero.

9.6 Hardware Effects

While I focused on results taken using *laserbay*[90] in this section, variations in hardware did have a significant effect on compression outcomes. For example, when performing a Comptool test at 100Mbit/s with Facebook data and no contention, *laserbay* is able to save 27% with “best” and nearly as much using lzop as a static strategy. At 1Gbit/s, *laserbay* can save 14.3% using “best”, and 8.2% using lzop as a static strategy. However, on *leap6* at 100Mbit/s (and 2.4Ghz), “best” saves only 23.5% while lzop is *indistinguishable from null*. At 1Gbit/s, the situation is even worse; “best” saves only 9.7% while lzop loses 168%.

Ultimately, because of *leap6*’s lower computational power, it must fall back to “lighter” compression Methods earlier as bandwidth increases (in order to maintain the necessary AOR), resulting in a larger proportion of input being sent uncompressed. For example, on *laserbay* at 1Gbit/s, null and lzop share the workload (in terms of Method choice counts) almost equally, while for *leap6*, only about 10% of events were compressed using lzop with the rest using null compression.

Hardware (or system software) can also affect chunk size selection when maximizing throughput. Figure 67 shows the different chunk size selections on *laserbay* versus *leap6* made while identifying the best strategy for YouTube data at our upper three bandwidths. *leap6* almost completely avoids making 512K chunk size selections, while it is the most common chunk size for *laserbay* in the same scenario. Since these chunks were chosen to be part of the “best” strategy based on their actual performance – which was executed on different machines –

[90] Descriptions of the test systems are included in Section 9.1.

it is difficult to identify the magnitude and cause of this effect. It could be hardware-based (e.g., CPU caching), kernel-based (e.g., caused by buffer settings which could differ by kernel and distribution), or software based (e.g., differences in the Python or other libraries used by Comptool). Regardless, it is important to recognize that static chunk size strategies may not be ideal for different systems.

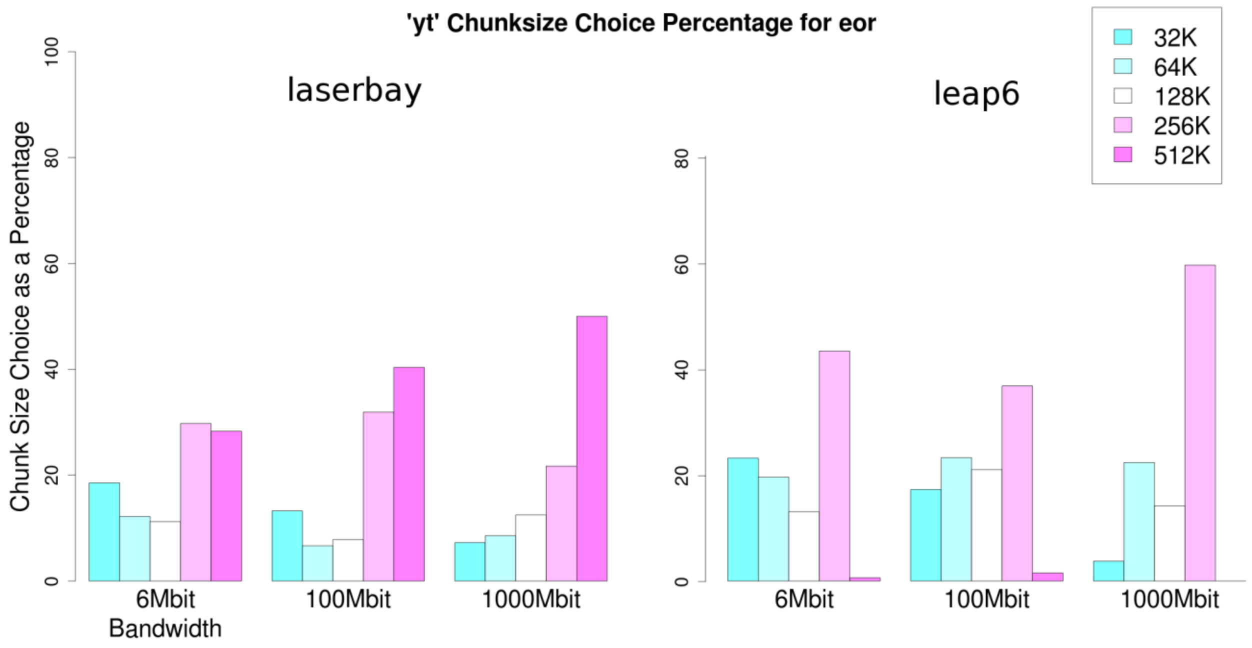


Figure 67. Effect of hardware characteristics on chunk size selection.

9.7 Discussion

The primary goal of Comptool is to facilitate the identification of "best" strategies for AC in terms of both throughput and space. To do this, Comptool identifies opportunities that can benefit from AC and estimates the scope of the potential improvements. In the process, Comptool identifies factors that determine whether and how compression can improve efficiency.

In the process of evaluating Comptool's capabilities using input sources for Datacomp, I have identified a wide variety of real-world data types that can be significantly improved using

AC, including popular websites and data types. For example, Comptool is able to improve throughput for Facebook data by 27% and 14%, and Wikipedia data by 52% and 28% at 100Mbit/s and 1Gbit/s, respectively. This goes against the conventional wisdom that compression is only useful at low bandwidths, on especially compressible data, or for receivers (Comptool emulates sender-side compression). Comptool has also identified data types that are difficult (or impossible) to compress.

Furthermore, the improvement identified by Comptool does not simply perform the status quo strategies of null compression or gzip-6. Instead, most types of data have their own "Method signatures" – mixtures of Method choices that are best for a given situation. In situations where static Methods are not successful but the data is still compressible, Comptool tends to use a mixture of null compression and LZO, a well-known high-speed compressor with only moderate compression. When data is uncompressible, Comptool correctly identifies that no compression is the most efficient choice.

Comptool shows that the best AC strategies can indeed depend on the combination of many different factors. The main dynamic (i.e., non-hardware) factors affecting throughput include the available bandwidth, data characteristics, and Methods used. Methods themselves are determined by the specific algorithm used in combination with the chunk size and strength level, all of which have been shown to have meaningful effects for AC.

While the unsurprising fact that raw computing power has been shown to be significant in the differences between *laserbay* and *leap6*, one surprising result is that CPU load and frequency do not make a very significant difference in terms of the ultimate best strategy. However, I feel that this is not so much a negative result as it is inconclusive. On the one hand, the fact that the contention workloads had so little effect on the throughput of the system suggests to me that they did not produce enough overhead to elicit significant system effects.[91]

Similarly, *leap6* only has two CPU frequency levels and the entire processor must be in one of those two states. This is not especially energy-flexible. Perhaps different workloads including a broader range of frequencies (and multi-threaded compression Methods) on a more flexible system would show larger effects. It's also possible that Comptool's nature as an "offline" AC tool might be artificially avoiding much of the timing issues created by computational resource issues. These are all questions for future work.

On the other hand, contention and CPU frequency *did* demonstrate an effect on compression Methods which were already computation-starved; it is just that improving computational resources did not improve these Methods enough to change their importance in the "best" strategy for the scenario. This is a valuable observation because it underscores the importance of the components in the EEOR calculation (AOR, ABW, and CR) when making AC choices. Combining this insight with more diverse tests including different frequency and contention workloads (rather than modulating bandwidth) could be interesting.

Another result provided by Comptool is that data types cannot be placed on a simple "compressibility spectrum" of easiest to hardest when throughput is the goal, because CR and AOR varies by compressor and environment (including both software and hardware factors). This is especially true because LZO, relied upon by AC strategies when computation time is short, has some significant CR degradation for certain data types, including binaries and email. Comptool also indirectly demonstrated that 512K chunks are, for some purposes, too small. Powerful compressors, such as bzip2 and xz, have compression ratios similar to more common

[91] This suggests that simple CPU utilization percentage is not the most useful metric, in that 100% use doesn't seem to be especially troublesome for performance.

compressors (e.g., gzip) when the input provided is between 32K and 512K. However, they can achieve tremendous compression gains with larger inputs.

These and other observations combine to support the observation that in the realm of AC, simplifying assumptions may be helpful, but they are imperfect and will likely result in undesired outcomes if the system is used in environments or on data that it was not designed or tested around. For example, ACE uses 32K chunks because they were empirically identified to be the best (by the researchers), even though Comptool's results show that, when available, other moderately larger chunk sizes may be more efficient for compressors like bzip2 (which was used in ACE). Even larger chunk sizes (in the megabytes) can significantly improve compressibility for heavy duty compressors, although this would come at an adaptivity cost.

ACE [62] and ACCENT [78] both use assumptions about linear performance relationships existing between compressors based on a small number of statistics, such as CR for a given input. These assumptions are used to build simple, but effective Models informing Method choice based on observed system state. However, Comptool's results suggest that these kinds of Models would not be as effective as possible (at least for the kinds of data, environments and quantization levels tested here), without (at least) the addition of separate Models for different types of data. This is because the relationships between the compression performance of different Methods can change significantly based on data characteristics. While these Models can change along with the data over time, doing this runs the risk of making mistakes while the Model is learning as the result of a change, or regularly making mistakes if the data changes faster than the Model can learn.

Finally, while Comptool serves its purpose admirably and produces reams of interesting information, it is not a real AC system. Most importantly, it does not pay for any of the expertise it appears to demonstrate, because the expertise is all performed offline. Comptool can show interesting and useful effects that can help researchers or developers, and it can be used to help validate real-world AC or compression systems. However, to be compelling as solutions, real AC systems must have a “profitable business model” – they must save more than they spend.

10 Drcp Results

Drcp is a simple socket communication wrapper around Datacomp. It is able to use any priority or mode supported by Datacomp, such as the space-maximizing Model, but is primarily intended to demonstrate Datacomp's ability to improve the efficiency of network throughput. Like dzip, drcp internally quantizes the input into smaller pieces on the fly, using Datacomp. This enables the application of a *series* of compression Methods (rather than being forced to use one Method on the entire input) in an attempt to maximize resource use. Where dzip attempts to maximize space savings, drcp attempts to maximize communication throughput. Since drcp is really a thin wrapper around Datacomp, an evaluation of its results is essentially an evaluation of Datacomp's ability to improve throughput using its "time/energy" decision Model. This Model uses a database of historical performance in conjunction with Datacomp's mechanism, Models, and Monitors.[92]

10.1 Goals

Dzip and Comptool's results (Sections 8 and 9) strongly suggest that quantization and greedy optimization result in choices that are as good as the best static strategy, but that the results of this process are usually not *better* than the *best static* Method for a particular opportunity (at least not in a statistically significant way). However, Comptool's throughput results demonstrate that there is no single compression Method that is best for all Opportunities. While the adaptive process employed by Comptool can always achieve the best-known throughput (or compression), doing this requires changing the compression Methods used. Furthermore, Comptool shows that using one Method ubiquitously for all Opportunities is not just sub-optimal, but that doing so will ultimately hurt performance.

[92] Data and other information regarding these experiments will be available at http://labs.tastytronic.net/datacomp/

For example, the results in Section 9.4.2 show that while gzip is the best performing strategy for Facebook data between 500Kbit/s and 6Mbit/s, LZO is the *single* best strategy at 100Mbit/s and 1Gbit/s. Looking at the individual compression Methods selected by Comptool at 1Gbit/s (Figure 37), it is apparent that Comptool's "best" strategy is actually composed of both LZO and null compression – using LZO alone would not have resulted in superior performance. However, the mean results (Figure 36) are not better than LZO in a statistically significant sense.

If beating the best static strategy is not generally possible with a greedy quantized approach, then drcp's best-case result is to always do as well as the best static strategy for the given opportunity. In this way, drcp would always "do the right thing," performing better than any single static strategy (e.g., always compressing with gzip). In other words, "success" for drcp means matching the most efficient static strategy for an opportunity, while "failure" means performing worse than the best static strategy.

10.2 Challenges

Quantization eliminates some potential throughput benefit due to smaller input sizes affecting compression ratio and increases in overhead per-byte. However, we know that this overhead is not overwhelming, because Comptool's throughput results (Section 9) show that it can consistently improve efficiency where we would expect it to be possible. However, while Comptool pays the cost of compression and transmission, it does not pay any "adaptation costs" (such as the costs related to decision making) or operational costs (such as reading files from disk or network) because Comptool's decisions are made offline and its mechanism is designed specifically to capture only the cost of compression and transmission. In contrast, drcp (or any other real AC system) must pay those costs.

Thus, in order to be profitable, Datacomp's time/energy Model must improve efficiency beyond the cost of these overheads. The Model's costs can be broken down into subcomponents, such as the cost of identifying the data type (i.e., bytecounting), monitoring the ABW, CPU load and frequency, and history database costs. Prediction error is yet another kind of overhead, resulting when the Model is incorrect or receives bad input from the monitors. Ultimately, the goal for an AC system is to maximize the benefit provided by each component while minimizing its costs.

In summary, Comptool identifies results that are theoretically possible, without actually achieving them, while Datacomp must achieve them in the real world. As a result, we would expect Comptool to normally outperform Datacomp both in terms of raw throughput and the optimality of its choices. However, Datacomp does have some advantages over Comptool that may help it recoup some of its costs: it supports parallelized compression, and it is written in a compiled language. Thus, evaluating Datacomp's ability to improve throughput as compared to Comptool is primarily about identifying how much Datacomp can improve efficiency and comparing that to the kinds of gains Comptool identifies as theoretically possible.

10.3 Experiment Design

To evaluate the ability of drcp to improve throughput, I performed experiments similar to those described in the Comptool results section (Section 9). Each discrete test measured the time to send one-megabyte "sample sets" of various data types over a communication channel while controlling the bandwidth and environmental parameters. Since drcp actually performs the network communication, no analysis phase was necessary; I simply averaged the resulting runtimes and calculated confidence intervals as previously described.

To train Datacomp's Time Model, I essentially created a large number of experiments using samples from the "training set" that was separated from the "testing set" early in the workload selection process. As described in Section 6.5.3.3, the History Database has a number of mechanisms to ensure that the database learns as quickly as possible (selecting unused strategies even if other good strategies have been identified, "boosting" underused options in case they were trapped in minima, etc.). I ran and reran those experiments until the database consistently reported that it had "no misses" during experiments – times when it read the database but found one or more Methods untested. Then, I performed all future tests using Datacomp's database in a "read-only" mode. In other words, during all testing with data from the "testing set," the history mechanism was barred from adapting further to the data. In this way, all the "knowledge" in the database was formed based on training data and Datacomp could not learn during the tests.

All drcp experiments were performed on *laserbay* using the same sets of sample data, bandwidths and CPU contention levels used in evaluating Comptool (see Section 7). While the CPU frequency was monitored and used for making decisions, unlike Comptool on *leap6* the CPU frequency was not experimentally controlled. Instead, it was controlled dynamically by the Linux kernel. Also unlike Comptool, drcp actually performed network communication over a dedicated 1Gbit/s Ethernet link between *laserbay* and a second host, *target* (which has nearly identical hardware, software and configuration).

Although the current Datacomp implementation includes fully-functional code to handle threaded reading and decompression, these experiments do not test decompression costs. Instead, data was accepted but discarded and timing was performed only on the sender side. This is for two reasons. First, as a standalone tool, Comptool does not measure decompression costs, so it would be harder to compare the results from Datacomp and Comptool directly. Second, while decompression is normally faster than compression for equivalent systems and Environments, this is not necessarily the case. Additionally, decompression cost can be significantly affected by the efficiency of the implementation. While I fully intend to perform decompression tests using Datacomp in the future (with asymmetric hardware and conditions), I felt that performing compression-only tests was a simpler and more transparent evaluation of the Time Model.

As discussed in Section 7.4.1, I use dummynet [109] on the *receiver* side to limit bandwidth. Receiver-side limits are used because, thanks to buffer mischief and other issues similar to the challenge of throttling pipe throughput (see Section 5.3.3), it can be difficult to accurately limit bandwidth on the sender side. By limiting throughput at the receiver, the sender-side mechanism is simply forced to obey the intended limit.

To emulate real-world uses of compression as closely as possible, these results compare the Time Model results with static compression performed by drcp using the Manual Model. This ensures that the costs of network transmission and the software itself are included; the only difference is the cost of the adaptation mechanism and the benefit of its decisions. In the static tests, drcp is configured to use null, LZO, and zlib-6,[93] each compressing the largest chunk size (512K) with a single thread.[94] Xz is not included because while it is a powerful compressor, it is too slow to be effective at higher bandwidths.

To make sure that the static modes do not pay unnecessary adaptation costs, drcp in manual mode does not pay the costs of using the history database, estimating the ABW, checking the CPU load and frequency, or bytecounting the data. I can then compare the proportional gains

[93] Zlib's strength level 6 is the default.
[94] Datacomp always uses separate sending and compression threads, even where it only uses one compression thread.

achieved by Comptool versus static compression (Section 9.4) against the proportional gains achieved by drcp using the Time Model versus drcp using static Methods dispatched by the Manual Model.

10.4 Throughput Efficiency (Time / Energy)

In this subsection, I show and discuss the results of drcp's performance for various file types when optimizing for time. The bar graphs show the difference between drcp's performance using the AC strategy "Time" and drcp's performance using the static strategies null, LZO, and gzip-6. Unless otherwise specified, the graphs show the results acquired with no CPU contention workloads and with dynamic CPU frequencies, controlled by the kernel on *laserbay*.[95] Differences are shown as a percentage of the time spent by null on the given task (similar to the graphs for Comptool shown in Section 9.4.). Confidence intervals were calculated using the standard mechanism for this project (see Section 8.3). Because these graphs represent the cost difference for each Method with respect to null, the "improvement percentage" for null is always 0%.

10.4.1 Zero

Figure 68 shows the percent change achieved by the Time Model over null compression when transmitting zero data using drcp. For 1Mbit/s and 6Mbit/s, all types are able to essentially double throughput (TP). At 100Mbit/s, LZO and "Time" are able to improve TP by only about 87%, while zlib's performance slips, improving TP by only 70%. At 1Gbit/s, "Time" and LZO improve TP by ~66%, while zlib is only able to improve performance by 11%.

[95] In general, CPU contention does have an effect and is therefore important to describe. However, the overall effect is straightforward and a result, is typically not shown.

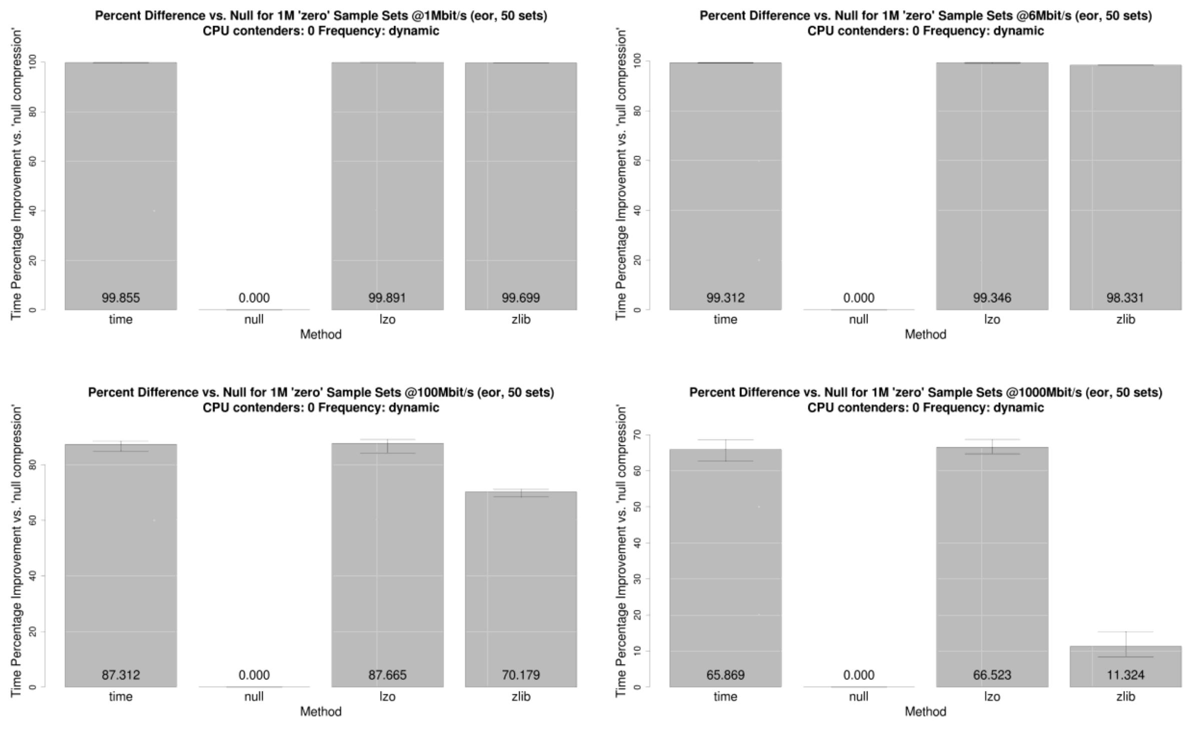


Figure 68. Percent change for zero data.

Comparing these results with Comptool's (Section 9.4.2), we find that drcp and Comptool achieve roughly the same improvement at lower bandwidths. However, at higher bandwidths Comptool is mostly able to maintain the same improvement levels (99% at 100Mbit/s and 96% at 1Gbit/s), while drcp's overhead diminishes its performance slightly (87% at 100Mbit/s and 66% at 1Gbit/s). This results in part because Datacomp's Time Model spends roughly the same amount of adaptation time on each byte. As bandwidths get higher, the relative impact of adaptation increases, regardless of benefit. Still, drcp succeeds according to our criteria (at all bandwidth levels) because it always achieves close to the best static strategy that could have been used.

As an aside, it would be nice to be able to see the specific choices used by drcp in the processing of this information, as we can for the Comptool results (Section 9.4). However, while Comptool can log every bit of useful detail during its processing, drcp cannot typically afford to log its choices during experiments due to the risk that it would negatively impact throughput.

However, the previous results of this project (in particular, Section 9.4) have shown that since Comptool does not typically beat the best static strategy, we can estimate the composition of adaptive Strategies by comparing the Results for the adaptive strategy against the results for other static strategies.[96] Based on the throughput means, it appears that "Time" uses LZO almost exclusively at high bandwidths.

10.4.2 Pseudo-random

Figure 69 shows the effect of drcp on pseudo-random data (generated by the kernel and read from `/dev/urandom`). At 1Mbit/s, "Time" pays a "statistically significant" 0.008% penalty for this data. This is not entirely surprising, because the Time Model has adaptation costs that null compression does not pay. On the other hand, 1Mbit/s is so slow relative to Datacomp that it is unusual to see any penalty at all. Regardless, even if that result is not noise it is small enough to be ignored.

Additionally, while it may be surprising to see LZO and zlib pay what visually appears to be a "heavy" cost relative to null at 1Mbit/s, it is important to recognize that this value is a percentage (2% and 1.5%, respectively) of null's runtime. Since null's runtime is around 8.0s (the cost to send 1M on a 1Mbit/s network), the actual overheads for LZO and zlib are between 0.12 and 0.16 seconds per megabyte, which seems realistic for the compressors given that the data is uncompressible. Incidentally, the fact that Time does *not* pay this penalty demonstrates that it is indeed recognizing and using null compression.

[96] Datacomp can write extensive logs of the choices made (and their costs) for any given experiment. However, building up a data set for each experimental environment – without interfering with the "timed" experiments – would require a complete second set of experiments that might have different results due to the instrumentation.

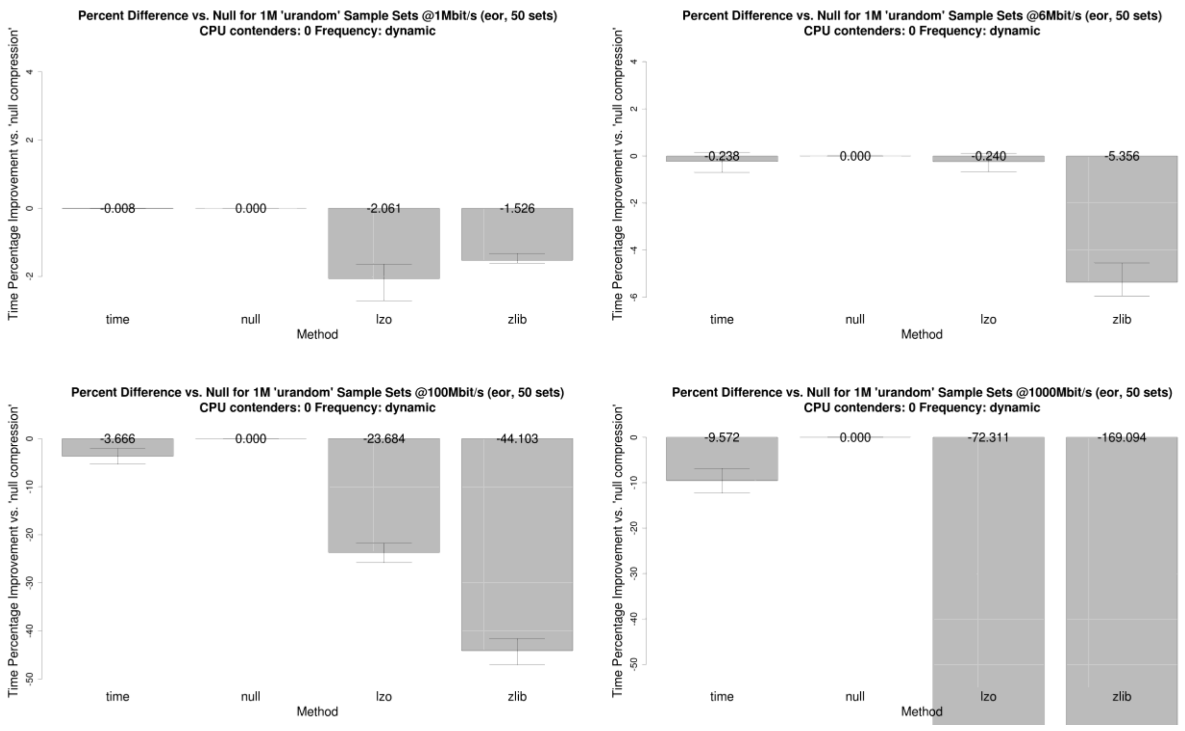


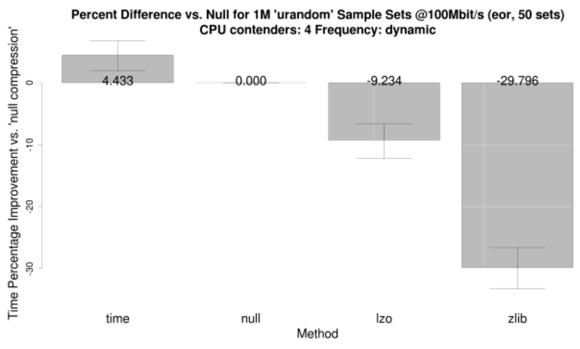
Figure 69. Percent change for pseudorandom data.

Performance for Time at 6Mbit/s is more like what we would expect; it is statistically indistinguishable from null, as is LZO. This illustrates two important factors at play concerning compression choices and results. As bandwidths increase, the bottleneck changes from being the network to being the execution Environment – the software and hardware. Because the software operates within a live multi-tasking operating system, this means that the variability of results tends to increase with the bandwidth. This can explain why LZO, which suffered a small penalty at 1Mbit/s, actually appears to have "improved" performance at 6Mbit/s (in that it is statistically equivalent to null). However, as bandwidths increase, penalties for algorithms that are too slow also increase. This means that we would not expect to see a Method have a significant cost at one bandwidth but then be profitable at a higher bandwidth. For example, zlib – a costlier compressor compared to LZO – suffers a worse penalty at 6Mbit/s than at 1Mbit/s.

At 100Mbit/s (lower left) the bandwidth is 17 to 100 times faster than the previous bandwidths (6Mbit/s and 1Mbit/s, respectively). We would expect this to exacerbate both the relative overhead of adaptation and the differences between the compressors (since this data is not compressible at all). This is in fact what happens. Time recognizes that the data is uncompressible, but at 100Mbit/s it pays approximately 3.7%, which is due almost entirely to the overhead of bytecounting.[97] Performance for LZO and zlib continue to degrade as they are unable to keep up with the throughput. Finally, at 1Gbit/s, Time's performance has degraded to 9.6% (again due primarily to bytecounting), but LZO and zlib have much larger losses, falling to -72% and -169%, respectively.

CPU contention, which makes a difference for drcp even though it did not have a significant effect for Comptool, can actually improve *relative* performance results. For 1Gbit/s and pseudo-random data, contention created by the script "bzip2urandom" (Section 7.4.2) increases the runtime of all strategies – including null – but the increase is not uniform. Runtime for "time," null, and LZO increase by approximately 0.06s, while runtime for zlib increases by almost 0.08s. This leads to distinct proportional improvements for each Method with respect to null compression. For example, where LZO's penalty at 100Mbit/s with no contention was approximately 24%, with contention it is reduced to approximately 9%.[98] The heavy load also increases the variance of the results, so it is more likely that confidence intervals will overlap.

[97] I determined this by performing a drcp test on the concatenated 50M corpus of all pseudo-random filesets with logging enabled. The amount of time spent in the bytecounting routine was 3.7% (0.16s from a runtime of 4.327).
[98] This effect is analogous to the way in which the proportional age differences between siblings change as they grow older. When my little sister was one, I was 6 times older than her. Now, I am only 1.6 times older than her.



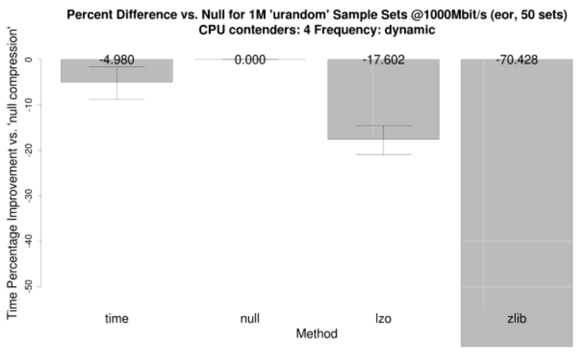


Figure 70. Effect of four contender threads on improvements for pseudo-random data.

This is shown in Figure 70, where (on the right) the loss imposed by “Time” on pseudorandom data at 1Gbit/s without contention (Figure 69) is reduced from ~9.6% with no contention to a loss of only about 5% with four contenders. The left side of Figure 70 shows the effect of contention at 100Mbit/s. Again with four contenders, “Time” now actually beats null (improving TP by approximately 4.4%), and LZO and zlib are improved significantly over their no-contention results at the same bandwidth (although they are still losing propositions). These results suggest that, all other things being equal, AC can improve efficiency for systems under heavy load. Again, this is because contention increases overhead, not because it somehow allows compression to succeed on uncompressible data. Subsequent graphs in this section show that this general effect applies beyond pseudo-random data.

Ultimately, drcp succeeds in matching the best case at 1Mbit/s and 6Mbit/s, but “fails” by my standard at 100Mbit/s and 1Gbit/s. This is not because drcp makes the wrong choice, but because drcp’s adaptation cost is too large to hide (losing 3.7% and 9.6% respectively). This happens primarily because the current Time Model pays a fixed adaptation cost *per byte*, which means that as bandwidth increases and test times decrease, a constant-length adaptation phase becomes a larger overhead that must still be “covered” with successful compression. That said, pseudo-random data is theoretically the worst-case data type. Datacomp is able to succeed at lower bandwidths (including the ranges of bandwidths that mobile users are likely to use) and adaptation costs could be reduced or made dynamic in future designs (see Section 11). This suggests that at the lower bandwidths, it would be difficult to "trick" Datacomp into wasting energy.

10.4.3 Wikipedia

Wikipedia data is the easiest to compress out of all realistic data types studied in this work. Accordingly, compression (as shown in Figure 71) significantly improves performance at lower throughputs. Furthermore, even though Datacomp struggles to keep up with gigabit rates in general, both Time and LZO are statistically indistinguishable from null compression at 1Gbit/s. Zlib is not so lucky; at 1Gbit/s, zlib loses 134%. This shows that, even though the data is quite compressible on average, it is not the case that zlib is always the best choice.

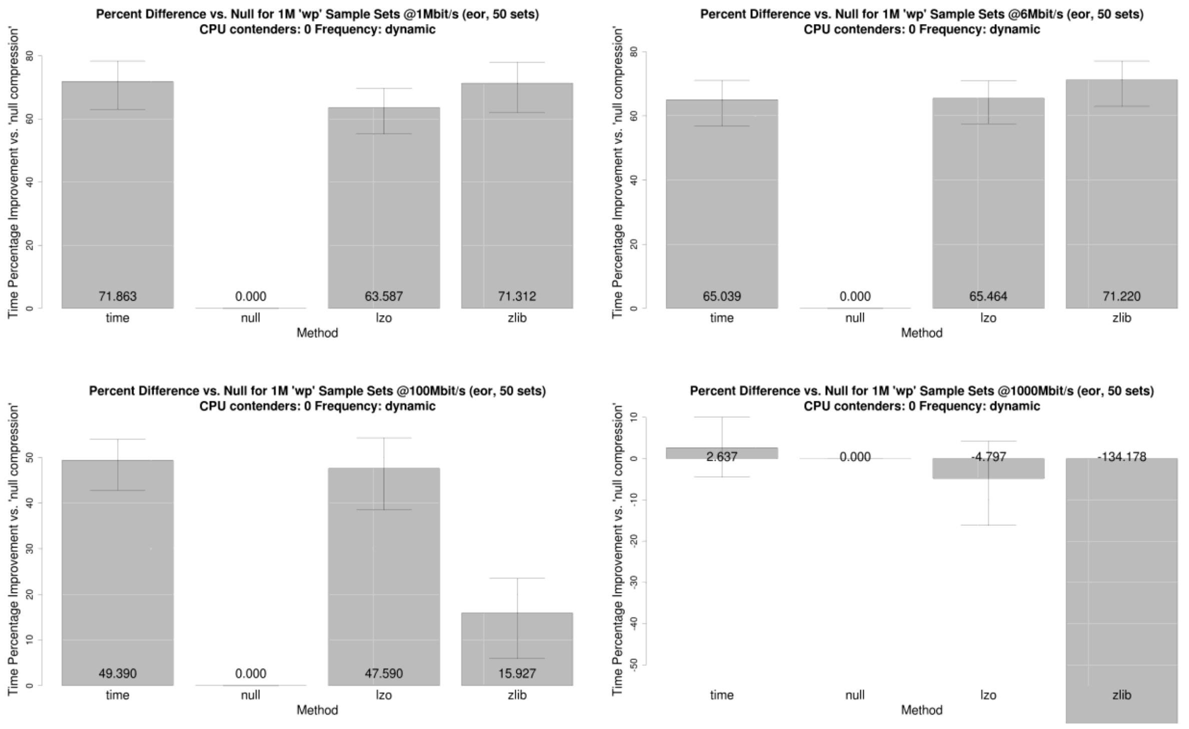


Figure 71. Percent change for Wikipedia data at various throughputs.

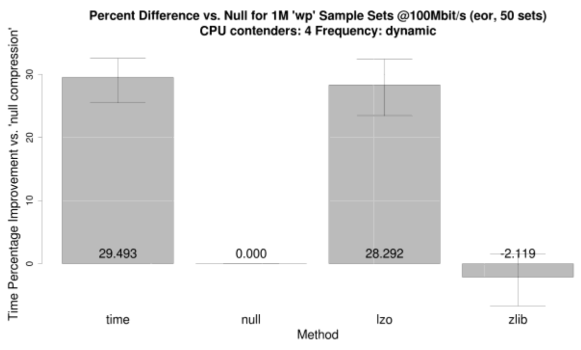
When comparing drcp's Wikipedia results against Comptool's results for the same environments (Figure 32), it turns out that drcp does quite well. For 1Mbit/s, 6Mbit/s, and

100Mbit/s, Comptool estimated improvements of 74%, 64% and 52%. For these bandwidths, drcp was able to improve performance by 72%, 65%, and 49%, respectively. In other words, drcp is able to improve efficiency up to within 1-3% of the best improvement Comptool was able to find using an offline greedy mechanism.

However, at higher bandwidths, drcp's adaptation cost (a cost that Comptool does not pay) becomes a larger fraction of the cost to send the data uncompressed. As a result, at 1Gbit/s, Comptool (which pays no significant adaptation cost) shows that the best strategy could improve throughput over null by nearly 28%, but drcp is unable to improve efficiency at that bandwidth. However, since Wikipedia data is highly compressible, Time is able to break even with "null" at 1Gbit/s. Thus, we can still say that drcp does as well as the best static strategy.

As with pseudo-random data, four contender processes change the outcomes of compression because contention slows the throughput of all Methods. Figure 72 shows the effect for 100Mbit/s (left) and 1Gbit/s (right). Here, the improvement provided by Time drops from 49% (with no contention in Figure 71) to 29% with four contenders, because LZO's performance (upon which Time's Strategy is based) is decreased. Also at 100Mbit/s, zlib ceases to be profitable with four contenders (although it was never the best choice for 100Mbit/s). This demonstrates that it is *not* the case that contention simply improves the results for Time or other Methods. Instead, Time's Results depend on the Results of all the Methods it could have chosen, the performances of which depend on many Data and Environmental factors.

At gigabit speeds with contention, "Time" manages to improve throughput by 6.6%. Again, this is not because contention somehow makes compression faster, but because it reduces the throughput of null, making Time more competitive. Nevertheless, this means that in the case of CPU contention, Datacomp is again more robust than using a single static strategy.



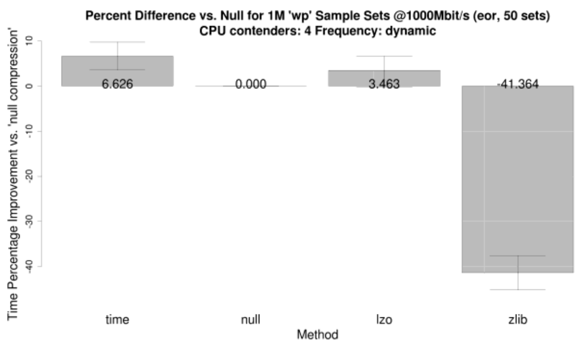


Figure 72. Effect of four contender threads on improvements for Wikipedia data.

10.4.4 Facebook

Throughout this project, Facebook data has served as special class of data, because it is well-known, not obviously easy to compress (compared to Wikipedia or plain text) and it comprises a meaningful portion of Internet traffic. Figure 73 shows drcp's performance on Facebook data in our standard setup. Time saves approximately 35% and 36% at the first two bandwidth levels. In both cases, Time's performance is statistically indistinguishable from LZO and zlib. Additionally, drcp's results are quite close to Comptool's estimate, which suggested that "Time" could save 34% at both bandwidths. For 100Mbit/s, zlib *costs* 20%, while Time *improves* throughput by 25% (Comptool predicted 27%), which is 9% better than the mean for LZO (the best static strategy), although the difference was not statistically significant. In other words, as with Wikipedia data, drcp's Time strategy succeeds in improving throughput at the first three bandwidth levels, achieving just 1-3% less than Comptool's greedy result.

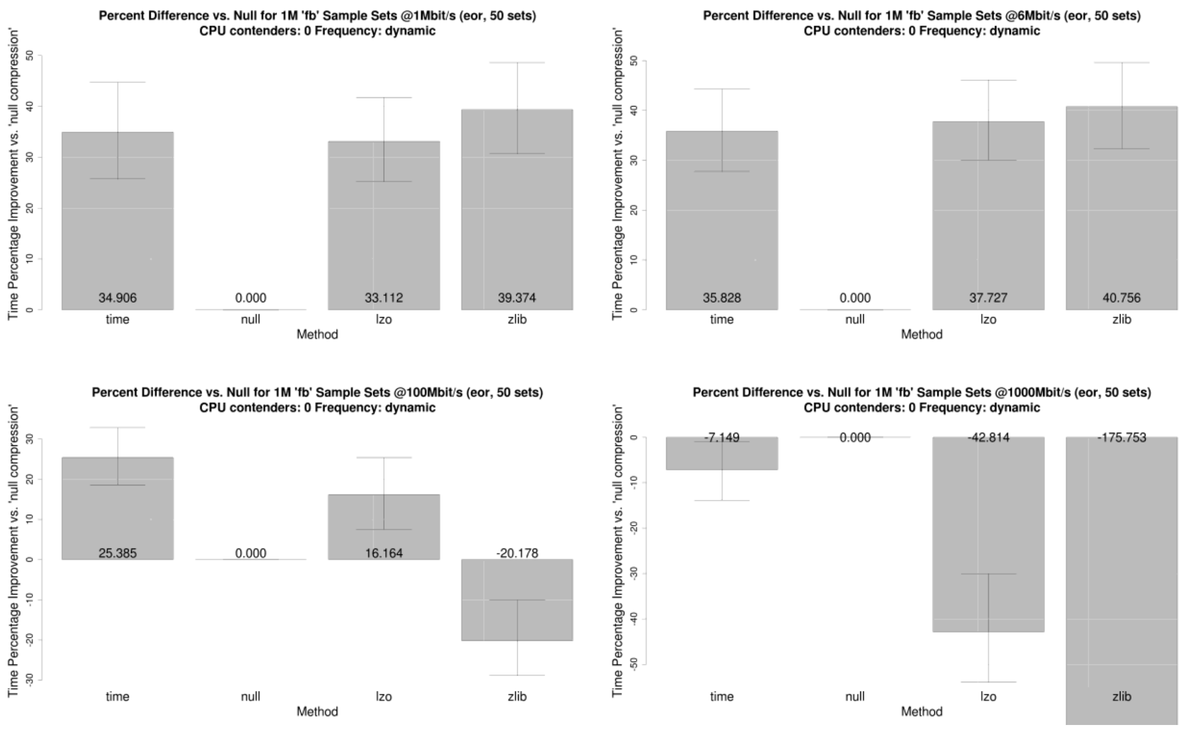


Figure 73. Percent change for Facebook data at four throughput levels.

At 1Gbit/s, Time loses 7%, although this is still much better than LZO (-43%) or zlib (-176%). Comptool estimated that "time" could save 14% at 1Gbit/s. While the poor performance of Time matches the adaptation overhead we have seen for other data types (up to 10%), the high cost of LZO for drcp as opposed to Comptool is surprising (where LZO saved 8.2% as shown in Figure 36).

There are several possible explanations for this. It could be due to the fact that the Comptool result is based on a single, whole-file compression without network overhead. In contrast, the drcp result includes program startup and shutdown, quantization, and network sends. At gigabit speeds, even tiny delays add up.[99] Finally, as with previous data types, four

[99] Another possible reason is that, while for Comptool the whole file compression was performed with the `lzop` utility, Datacomp's "default" LZO level (LZO1B) uses a slightly different algorithm than `lzop` (LZO1X). Both algorithms are included in the standard LZO library. This choice was made in an attempt to make the "fast" strength

contention threads comparatively improve Time's results. With four contenders at 1Gbit/s (not shown), Time's results are indistinguishable from null, although LZO and zlib are still wasteful.

10.4.5 YouTube

In the same way that Facebook data is our "average" type, YouTube data is one of our "hard" types (which, unlike random bytes is actually real-world data). While YouTube is technically slightly compressible, compressing it does not save much and comes at a time cost. As a result, we do not expect to save resources with drcp on this data. In fact, at 1Mbit/s, Time is equivalent to null, which is the best static strategy as shown in Figure 74. However, Time saves 2.7% at 6Mbit/s, making it better than the best static strategy. For both 1Mbit/s and 6Mbit/s, LZO and zlib's confidence intervals overlap 0 and are thus not significantly different from null. At 100Mbit/s, Time again "breaks even" with null, while LZO and zlib both cost (21% and 44% more than null, respectively). These results are all within the ranges predicted by Comptool. Thus, we can say that drcp succeeds at the first three bandwidths.

for LZO the fastest algorithm, with the "default" strength being slightly stronger and slower. However, it meant that Datacomp's "fast LZO" is the same as `lzop`'s default, possibly leading to this discrepancy.

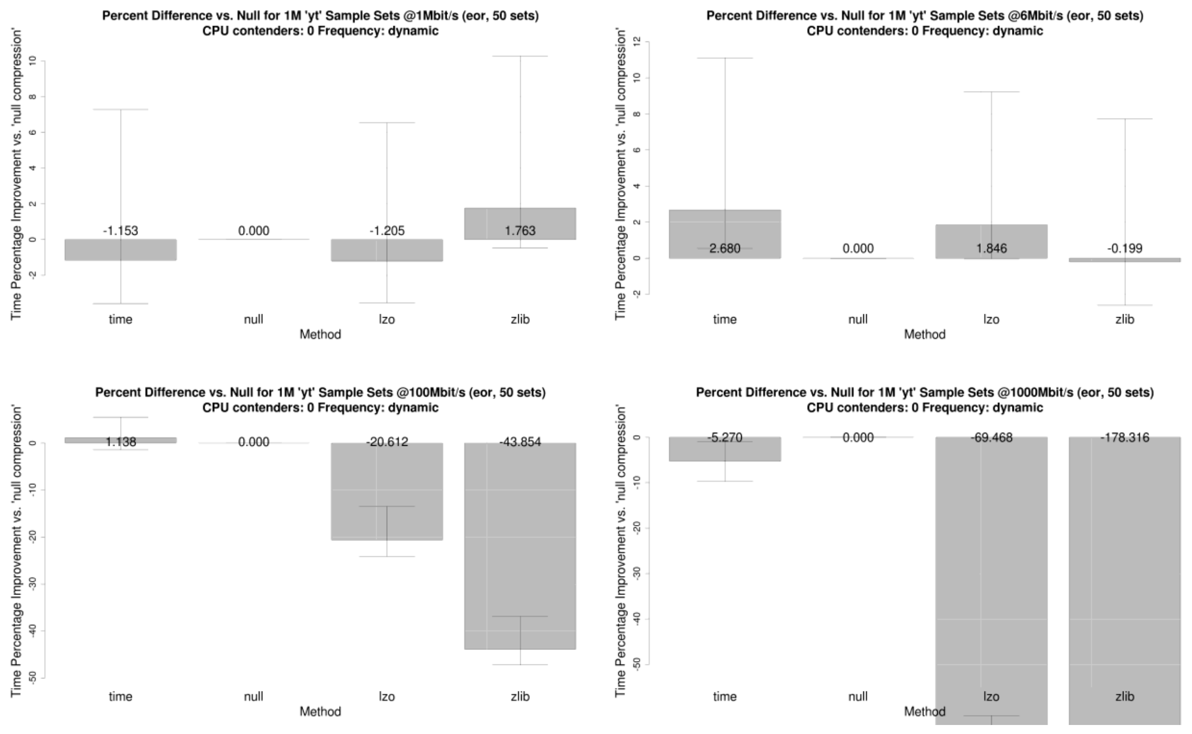


Figure 74. Percent change for YouTube data at various throughputs.

At 1Gbit/s, performance degrades for all compressors as expected. Time loses 5.3%, LZO loses 69% and zlib 178%. Thus, we can say that drcp does not succeed at 1Gbit/s, although this loss fits into a developing pattern. Interestingly, Time does *not* improve with four contention levels, but actually falls to -7.3% even though LZO improves to -16% and zlib improves to -77% with contention. This is interesting because it highlights that Time cannot be counted on to always improve with CPU contention.

10.4.6 Binaries and Mail

Binaries and email are interesting classes of data for AC. First, they are extremely common, and can include both compressible and uncompressible data. Second, although they are compressible by many algorithms, LZO – Datacomp's fastest compression option – is worse than other compressors on *both* types (and much worse in the case of mail) in a statistically significant sense (as seen in Section 8.4). As a result, they are a peculiar kind of challenge for AC. They're compressible, but their compression characteristics are apparently distinct from other types of data, at least for some compressors.

Figure 75 shows the performance of drcp on binary data. For both 1Mbit/s and 6Mbit/s, Time and zlib are statistically similar, although zlib's mean is slightly higher (about 60% versus 55%). At 100Mbit/s, Time's improvement is reduced to 32% (where "Time" and LZO have similar performance) and zlib has become a loss (-18.5%). These values also match Comptool's predictions fairly well (as shown in Figure 51), so we can say that drcp is successful for binary data on our first three bandwidths.

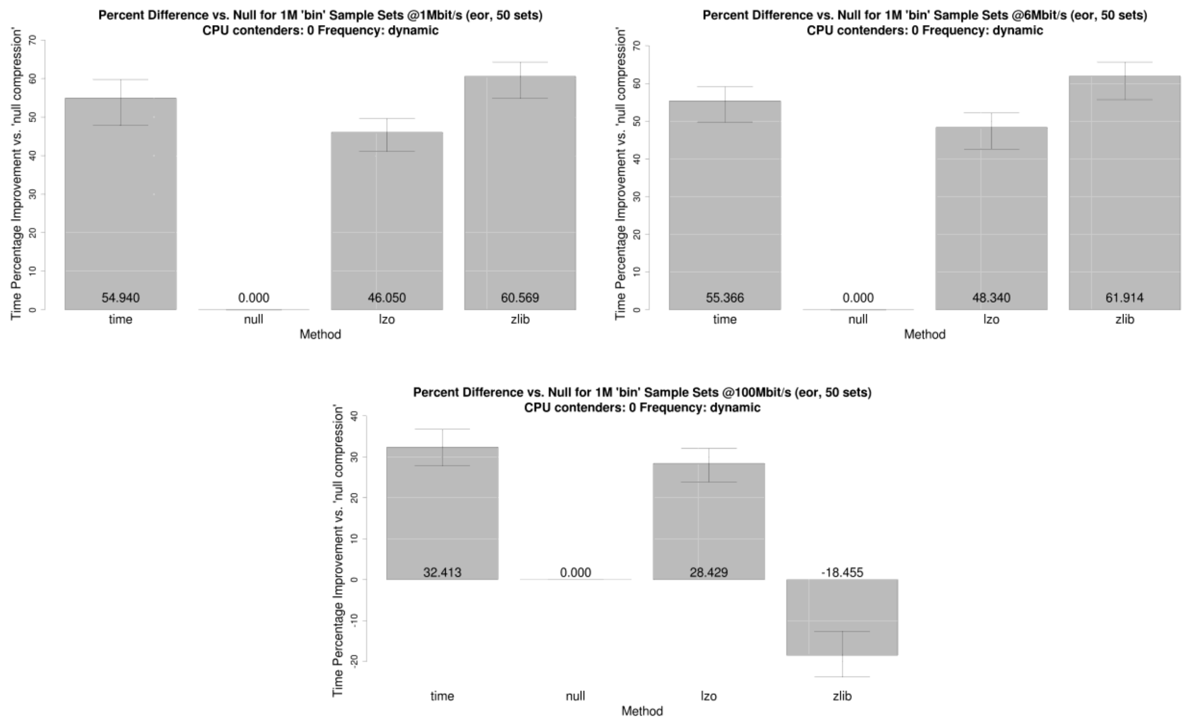


Figure 75. Percent change for binary data at various throughputs.

Before discussing performance for binaries at 1Gbit/s, it is useful to first show mail's performance at the lower bandwidths. Figure 76 shows drcp's performance on samples of mail. Performance at the first three bandwidth levels is good, but the results are different than for previous types, where LZO typically fared well at lower bandwidths.

For mail at 1Mbit/s, Time saves 30% where zlib's saves 34%, although the difference is not statistically significant. At 6Mbit/s, Time improves the result by 33% while zlib improves it by almost 36%, although their confidence intervals still overlap. Surprisingly, even though binary data is generally compressible, at 1Mbit/s LZO only manages to *break even*. At 6Mbit/s, LZO manages to save some TP, but only 5%. This is shocking since LZO is typically the fastest static strategy for compressible data. At 100Mbit/s, Time saves 10.4%, and beats any static strategy where both LZO and zlib are costly. Thus, we would say that Time does as well as the best static strategy at 1Mbit/s and 6Mbit/s and that it succeeds significantly at 100Mbit/s by beating the best static strategy.

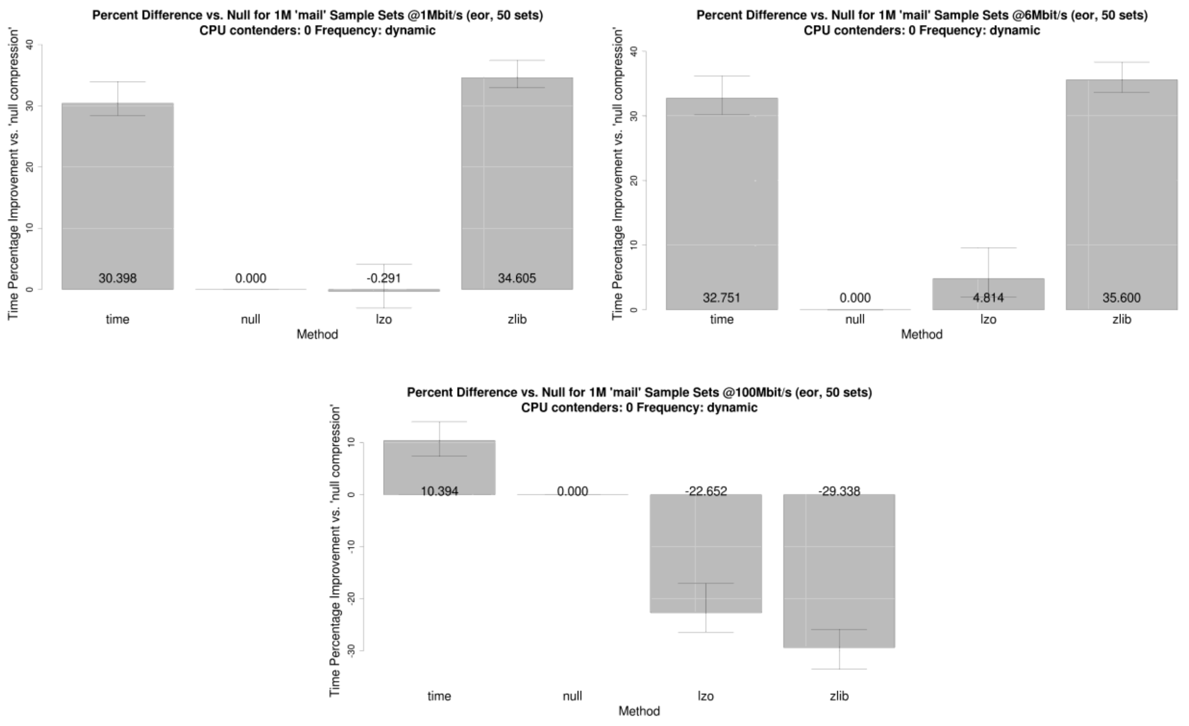


Figure 76. Percent change for mail data at various throughputs.

There are two anomalies here: LZO's general failure and Time's success of 10.4% at 100Mbit/s. The win for Time is approximately 7% better than Comptool's prediction (3%), and soundly beats the other static strategies. At 100Mbit/s, LZO results in a loss of 22.7%, while zlib results in a loss of 29%. The reason that Comptool could not predict Time's Result is that Time used a mixture of zlib-1 and zlib-6 (compressing chunks of the same input with one or the other Method), using two threads in parallel.[100] Comptool does not support parallel compression, and so this result was not available for use in its greedy solution. Thus, this result actually serves to show the benefit of parallel compression in Datacomp.

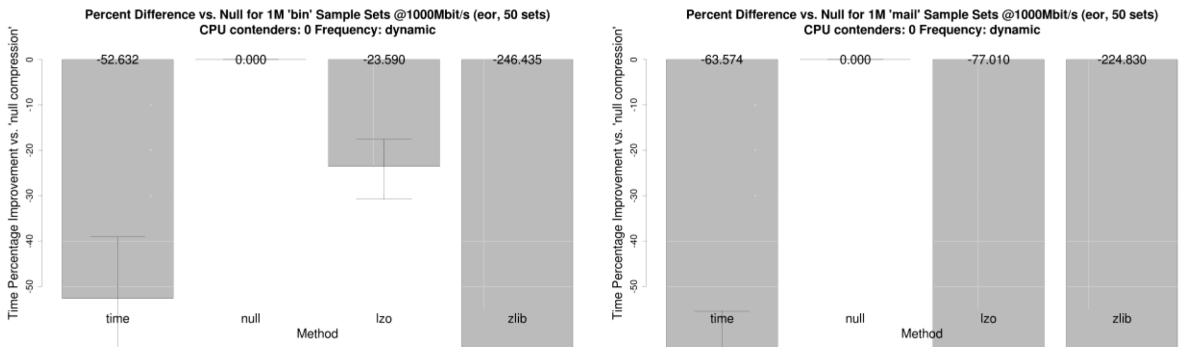
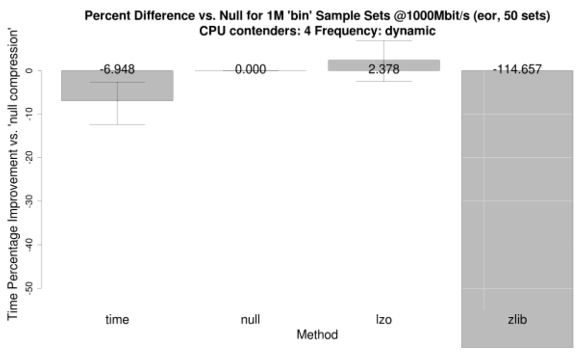


Figure 77. Performance improvement for binaries and mail at 1Gbit/s.

Investigating LZO's strange results leads us to the results for binaries and mail at 1Gbit/s. Figure 77 shows that Time makes serious prediction errors at 1Gbit/s for both binaries (left) and mail (right). At 1Gbit/s, Time costs 52.6% more than null, the best static Method. In contrast, Comptool predicts that the best strategy should improve performance by 11.3% (Figure 51). For mail, Comptool predicts a best result of 1.6%, while Datacomp loses a whopping 64%.

It is not surprising that drcp and Datacomp were unable to improve efficiency at 1Gbit/s. We know from previous data that adaptation overhead at 1Gbit/s typically costs 1-10% and that the best-case scenario for Datacomp at 1Gbit/s is to break even (when it is processing easy to compress data, such as Wikipedia). This indicates that the losses at 1Gbit/s are resulting from a genuine and consistent misprediction – not from normal sources of overhead.

[100] This was determined by running several experiments with logging enabled in drcp and manually investigating the Method choices made by the Time Model.



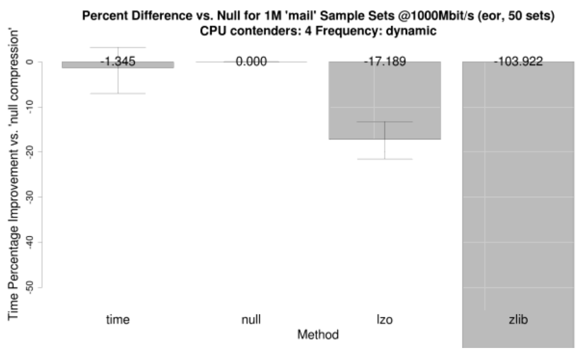


Figure 78. Effect of contention on drcp performance for binaries (left) and mail (right) at 1Gbit/s.

Adding to the mystery, the addition of contention levels at 1Gbit/s restores sanity to the values. As shown in Figure 78, performance on binaries is rehabilitated to its "standard sub-standard" performance – a loss of 1 to 10%, while mail fares better, with results that are statistically indistinguishable from null compression. Clearly, there is a reason why the Time Model is choosing to apply the wrong type of compression at gigabit rates without contention, but makes the right choice when the CPU is loaded.

There are at least two reasons for this failing. The first is that the room for prediction error shrinks by roughly a factor of 10 between each bandwidth level. In fact, my results show that Datacomp's adaptation overhead is too costly to succeed at 1Gbit/s unless the data is sufficiently compressible. This means that even if the data is compressible (like binaries or mail) the fastest compression mechanism is often not fast enough to succeed at a given bandwidth, because it is not compressible enough to cover adaptation. And while choosing the wrong compression Method at a low bandwidth might be imperceptible, at a high bandwidth it can add up to a significant proportional penalty.

For example, Figure 79 shows the absolute time for binaries (left) and mail (right) at 1Gbit/s. In both cases, Time takes about ~0.035s while null takes ~0.022s. For reference, on *laserbay*, an average call to the LZO library takes approximately 0.005s – so it does not take many mistakes (even if the "mistake" is to use LZO) to add up to a 0.013s error.

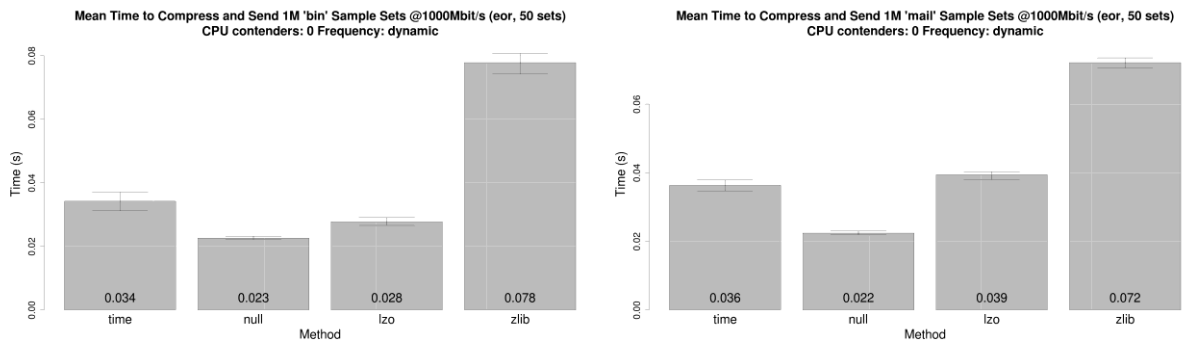


Figure 79. Absolute times for drcp on binaries (left) and mail (right) at 1Gbit/s.

The second reason for these maladaptive results is that binaries and mail "buck the trend" of the bytecounter, which leads to poor choices. They have statistically similar properties to types of data that benefit from compression at 1Gbit/s when the CPU is idle. However, they do not have the same performance properties when compression is executed. While the negative effects of this "aliasing" are not very visible at lower bandwidths (although they result in the poor strategy choice for mail at 1Mbit/s), there is no room for error at gigabit rates. Furthermore, since Datacomp was barred from "learning" during the drcp tests (Section 10.3), it would never be the case that the history mechanism would learn to "cope" with these issues.

10.4.7 Discussion

In practice, Datacomp's Time Model performs very well for all data types at 1Mbit/s, 6Mbit/s and 100Mbit/s. Its performance is usually statistically similar to the best static strategy, although in some cases it is significantly better than any static strategy by virtue of using a mixture of one or more Methods. Additionally, Datacomp's results at these bandwidths are typically very close in terms of proportional improvement to the analytically derived "greedy best" Comptool values. In some cases they are superior to these values, because Datacomp has two features that Comptool lacks: pipelined compression and transmission, and parallel compression.

The current prototype suffers from two main problems. First, performance at gigabit rates is rarely profitable, because the adaptation overhead alone costs more than uncompressed transmission. As a result, Datacomp loses 1-10% at gigabit throughputs unless the data is extremely compressible (e.g., Wikipedia or zero data). For patently uncompressible data (e.g., pseudo-random data), Datacomp also loses a small amount of efficiency at 100Mbit/s. The simplest way this problem could be fixed is a cutoff threshold for bandwidth, above which bytecounting would be suspended and compression would fall back to null.

Second, Datacomp occasionally makes poor compression choices for certain data types (i.e., binaries and mail), which results in significant costs at gigabit rates. This occurs because the bytecounting function is not perfect. In conjunction with the history database, the bytecounter conflates multiple data types that have similar statistical properties but significantly different performance properties. As a result, the database identifies compression Methods that reduce size but (for some data types) are significantly too slow for gigabit networks. Normally, if data was consistent enough in terms of type, Datacomp would learn that a strategy was failing after a few failures (the rolling average only "counts" the last 20 events). However, Datacomp's learning mechanism was set to "read-only" during the tests (see Section 10.3) to keep it from learning the test data.

In my tests, these issues were only serious at gigabit speeds. These rates are not practical rates for most mobile devices using wireless radios, including laptops and smartphones. Furthermore, the prediction error could be trivially overcome with a cutoff as just mentioned, or perhaps might be more dynamically overcome using additional bytecounting quantization levels, a different compressibility estimator, or some combination of the two. I chose not to pursue these approaches because I found the former to be "kludgey" and the latter to be out of scope.

While it may seem counter-intuitive, CPU contention actually improves rather than degrades the performance of Datacomp's time-optimizing Model in many circumstances. This is mostly because heavy system load drives down the throughputs of static choices and creates resource contention. These conditions provide an opportunity for parallelism and adaptation to succeed, even in scenarios where they otherwise could not, such as gigabit environments.

In the end, even though it is not perfect, Datacomp's Time Model accomplishes its goals. It significantly improves runtime and energy efficiency by matching or improving upon the best static strategy for a broad range of data types. And it does so without relying on external support or a significant amount of hard-coded human expertise.

10.5 Bytecounting and Database Access

The goal of this research is not to develop the most efficient compression performance estimator. However, the design of Datacomp's Time Decider Model depends on an estimator which is sufficiently efficient and accurate to be useful as an AC monitor. This means that the estimator should be inexpensive compared to compression and that it should be able to provide whatever type of information the AC Model requires. Since the estimator is meant to differentiate between different "compressibility types" (even if, from a human perspective, they are the same kind of data), it should provide a range of outputs to be used by the Model.

The value emitted by the estimator does not need to be the compression ratio. Additionally, while the levels of Datacomp's bytecounter can be *thought of* as representing a tier of compressibility levels (i.e., highly-compressible, compressible, minimally-compressible, not compressible), Datacomp's Model does not assume that there is an ordered hierarchy. Rather, it simply assumes that inputs of the same type have similar compression performance properties.

In this section I provide results showing the relationship between CR and bytecount over a broad range of data used in this research. This includes examining CR vs. bytecount for all data and by various types. I also discuss the issues that result in mispredictions by the Time Model for mail data in unloaded gigabit environments.

10.5.1 Bytecount vs. CR – All Types

Figure 80 shows a scatterplot of the gzip-6 compression ratio versus bytecount for every 32K chunk of all data types (except for the extrema of zero and pseudo-random data, since these have virtually singular bytecounts and CRs). There are approximately 11,000 data points; multiple points with the same CR and bytecount may be hidden by overplotting. The dotted vertical lines at x = 33, 66 and 100 show the division between data type tables in Datacomp's Time Model. The blue line and gray fitted model is a "Loess Curve"[101] and is based on all the data points (including any overplotted points).

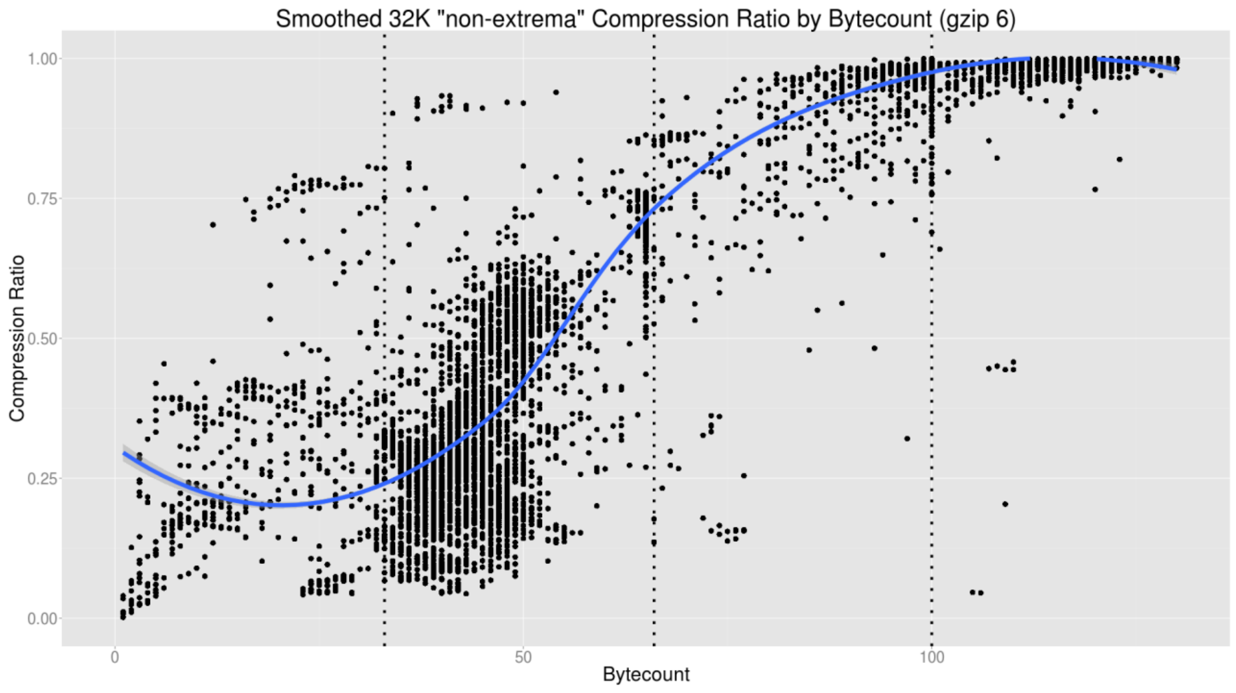


Figure 80. Compression ratio (for gzip-6) by "bytecount" for all 32K chunks of test data (except zero and urandom). Vertical lines denote time_decider() data type thresholds. Loess Curve shows hypothetical model relating CR and bytecount.

A number of high-level observations can be made from studying Figure 80. First, data with a bytecount above 100 is very difficult to compress. This is why Datacomp treats data with $b > 100$ as uncompressible. Data with bytecounts between 66 and 100 are variably

[101] This curve was generated by R's ggplot2 package using the defaults (which include a 95% confidence interval).

compressible, with the curve sloping fairly gently down to a CR of approximately 0.75. Most compressible data points falls between *b = 33* and *b = 66.* The bulk of these data points have CRs ranging from around 0.05 to approximately 0.60, while the fitted curve predicts CRs between 0.25 and 0.75. Finally, the majority of points where *b < 33* have CRs below 0.50 with the fitted curve estimating CR values around 0.25. To generalize, data in the lowest bytecount range is the most compressible, followed by bytecounts between 33 and 66, next followed by bytecounts between 66 and 99, and finally data with bytecounts above 100 are largely uncompressible. Clearly the largest spread of CRs is in the "middle" range between 33 and 66.

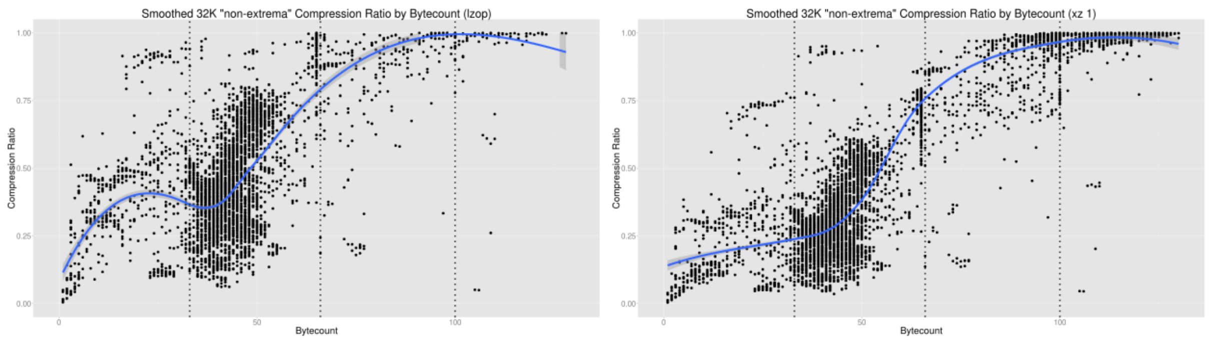


Figure 81. Compression ratio versus bytecount for lzop (left) and xz-1 (right).

Figure 80 shows compression ratios as calculated by gzip-6, however, we need our bytecount mechanism to be coherent for other compressors as well. Figure 81 shows the same type of graph, but with compression ratios as calculated by lzop (left) and xz-1 (right). There are differences; for example, xz-1 and gzip-6 (Figure 80) have a cluster of outliers around the coordinates b = 25 and CR = 0.75. However, for lzop the same cluster has a worse CR.

In fact, lzop seems to have meaningfully worse CR than gzip-6 and xz-1 on many chunks in the lowest range, resulting in the visible "bump" in the fitted model. These differences are not problematic, but simply indicate that lzop and gzip are Methods with discrete performance properties. In some Opportunities, gzip will be preferred because of its compression ratio, while in other Opportunities, lzop will be preferred because of its speed. In any case, our general observation – that CR increases from lower to higher bytecounts (and thus the different groupings have broadly similar characteristics) still stands.

10.5.2 Bytecount vs. CR – Specific Data Types

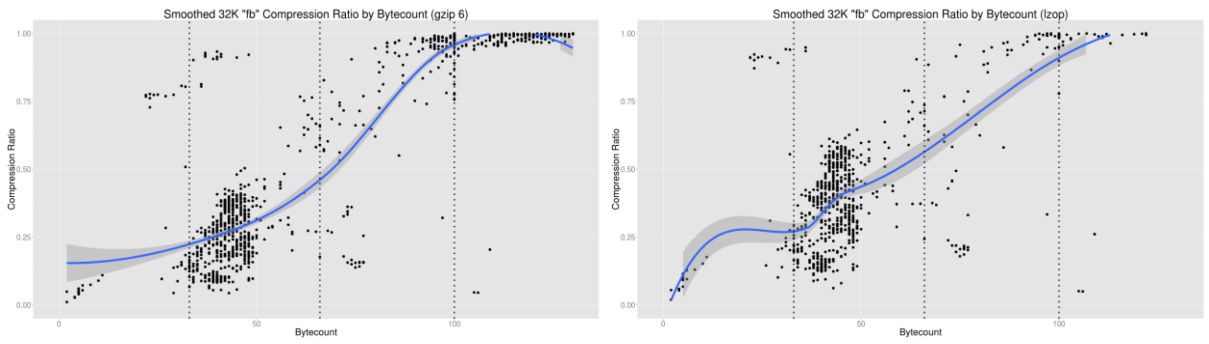


Figure 82. Comparison of CR vs. bytecount results on Facebook data for lzop and gzip-6.

Figure 80 and Figure 81 showed the bytecounts of all 32K samples vs. the CRs for specific compressors. Figure 82 “drills down” further, showing only the points relating to Facebook data for gzip-6 (left) and lzop (right). Like Figure 81, these curves are broadly reminiscent of each other but are not exactly the same, suggesting that Facebook data has generally consistent – but still distinct – performance Results for different Methods.

Perhaps the two most visible features are the strong clusters between approximately $b = 33$ and $b = 50$, and the cluster centered on $b = 100$. CRs for both clusters are similar between gzip and LZO, although gzip does generally better. The CR for the first cluster ($b = 33$ *to* 50) is between approximately 0.15 and 0.25, while the samples in the other group are largely uncompressible. Regardless, the general trend of decreasing CR from right to left holds for Facebook data.

However, not all data types fit the trend so well. Instead of Facebook data, Figure 83 shows the CR-to-bytecount plot for binaries using LZO. It shows that the bulk of binary samples are in the “middle” bucket, with CRs between 0.50 and 0.75. Comparing the samples in Figure

82 to those in Figure 83, it appears that Facebook data in this bucket is generally much more compressible than the equivalent binary data.

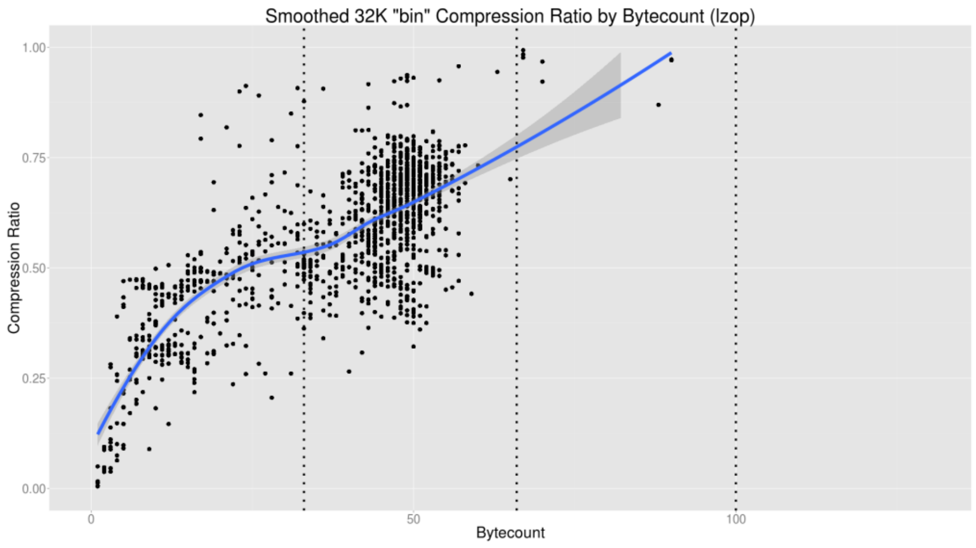


Figure 83. Smoothed CR to bytecount plot for binaries.

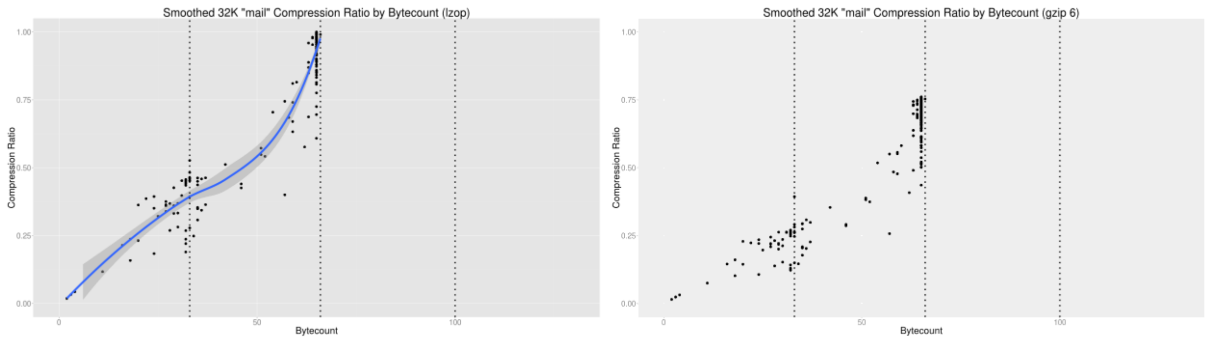


Figure 84. Compression ratio for lzop (left) and gzip-6 (right) vs. bytecount 32K chunks of mail data. The loess software is unable to fit a curve to the gzip data.

Some data types are even stranger than binaries. Figure 84 shows email compressed with lzop (left) and gzip-6 (right). Two things are immediately striking about this data. First, *no sample has a bytecount above 66*. Second, while gzip-6 is able to compress most samples to between 0.50 and 0.75, lzop is terrible at compressing samples in the same region, with many CRs well above 0.75 – even around 1.0.

We know that email is composed of smaller, compressible messages and larger, primarily hard-to-compress attachments. We also know that lzop is not a very effective compressor for many kinds of data, so it is not surprising that it would perform poorly for email attachments. What is surprising is that we would expect the hard-to-compress attachments, like JPEGs, to have high bytecounts – above $b = 66$ at the very least – but *no* mail samples have high bytecounts. Clearly there is something unique about mail data.

The answer to this mystery is that binary attachments for email are commonly sent and stored as base64-encoded data [115]. Base64 encoding is a scheme whereby an arbitrary stream of 8-bit bytes is transcoded into an alphabet of 64 printable ASCII characters.[102] Base64 encoding originated because many communication channels could not reliably handle non-printable bytes, and roughly half of the unique 256 bytes are either non-printable or are not standardized between systems. By encoding arbitrary data using bytes that were guaranteed to be represented properly, base64 served as a reliable interchange format. It also enables text and binary data to be stored in the same "plain text" file (as in email). It is for these reasons that base64 encoding (through the MIME and SMTP standards) became a standard component of email.

As a result of this unique encoding scheme, base64 email with a typical attachment will never have a bytecount much larger than 64, even though there may be information in the attachments that would normally not be compressible. This is because even though there may be non-base64 punctuation or other characters in the body of the email, the sheer volume of base64 symbol bytes will "drown out" any other bytes during bytecounting. However, because base64 encodes three arbitrary bytes into four printable bytes, it expands data by a factor of about 1.25

[102] The characters chosen are implementation dependent. On my system, they are [A-Za-z0-9.+] along with '=' to represent padding as necessary.

without increasing the amount of information. Because of this expansion, base64-encoded data is compressible, regardless of the content. Unfortunately, not all compressors are effective at recovering this space. In particular, Figure 84 shows that lzop is not able to compress this data successfully.

This brings us full circle to the problem discussed with binaries and email in Section 10.4.6. Drcp performed significantly worse than usual on binaries and email in 1Gbit/s environments with no CPU contention because the history model had learned that LZO was the best strategy for data in the "middle" bytecount bucket at 1Gbit/s without CPU contention. And in fact, that strategy works very well for Wikipedia data, which is able to "break even" with null compression at 1Gbit/s (as shown in Figure 71). However, the strategy works badly for binaries and email, because LZO does not compress them well and there is no room for error at 1Gbit/s.

Adding CPU contention improves binary and mail performance specifically because the heavy CPU load *discourages compression*. Digging into the history database values (not shown) indicates specifically that LZO at 1Gbit/s with no contention was identified as best for those types of data, but that null compression was best for similar Environments under CPU contention.

10.5.3 Discussion

This issue perfectly illustrates the potentially large differences in results between Methods when compressing the same data, and the dangers of assuming a relationship. Normally, if gzip can compress data, lzop can also compress it, though not as well. However, here is an example where gzip can compress data (email with a bytecount around 64), but lzop almost completely fails. In other words, it is dangerous to assume that the compression ratio achieved with gzip guarantees anything about the compression ratio for lzop or vice versa.

Significant differences between Methods can be good for a predictive model, because it offers real alternatives. However, if the results of using a Method on data of a specific type are unpredictable, then it can result in serious prediction error. For example, the samples for all data types as compressed with lzop in Figure 81 show that much of the data in the bucket between *b = 33* and *b = 66* is compressible with lzop, some of it extremely compressible. However, we know from Figure 84 that mail with a bytecount of *b = 66* is not very compressible at all.

As the database is trained, successful compressions with lzop improve the mean CR for lzop in that range, while compressions with mail degrade the CR. This can lead to decisions to use lzop when it will not be successful, as discussed in Section 10.5.2. Looking carefully at Figure 81, retrospect suggests that it may have been better to make the cutoff for the "middle" bytecount group just below *b = 66* – in other words, moving the hard-to-compress email samples into the next bucket. However, this kind of manual tuning is just as likely to cause problems somewhere else. In fact, I chose equal divisions to avoid hard-coding my own intuition into the system.

This general issue also highlights why using sample compressions with a fast Method (such as lzop) as an "efficient estimator" may not lead to the best compression decisions. "Fast compressors," which are not simply optimized versions of "normal" compressors, cut corners to achieve speed (otherwise all compressors would be fast). As a result, their performance cannot always directly determine the performance of stronger (if slower) compressors. In other words, using LZO to sample mail data would indicate that it was uncompressible regardless of the bandwidth, even though zlib could improve throughput at many bandwidths.

Finally, base64-encoded email serves as a prime example of data that would benefit from Datacomp or some other AC system. It is a real-world data type that is surprisingly compressible due to its encoding and the format or protocol cannot be easily changed because of its legacy. An AC system could transparently compress and decompress this data, reclaiming most of the space that was wasted without involving the user or changing the protocol.

10.6 Effect of Parallelism

The use of parallel compression can improve the throughput of specific Methods and thus help an AC system be more resilient to CPU contention. Datacomp supports parallelism as part of its mechanism (described in Section 6.3.3) by managing the compression threads individually. Datacomp will cut an input into pieces, compress each piece in parallel (using one to four threads), and send the pieces in order. In this way, Datacomp can perform parallel compression even if the underlying libraries do not explicitly support it.

Datacomp's Time Model automatically requests multi-threaded compression when the history mechanism predicts that it would be the most efficient choice. Manual inspection of the history database shows that threaded Methods are commonly, if not usually, selected over their single threaded counterparts. As a result, parallelism is one of the features that allows Datacomp to recoup its adaptation costs and in some cases enables Datacomp to find improvements beyond Comptool's estimates. However, because it is difficult to "see" the effect of parallel compression in the results already discussed, in this section I evaluate and discuss the performance of Datacomp's parallel compression facility.

10.6.1 Experiment Design

To explicitly test the impact of threading, I performed a number of tests using drcp with the Manual Decider configured to set the compression Methods directly. These parameters include the number of threads used, the chunk size, file type, Method, and contention level. The tests were performed between *laserbay* and *target* (described in Section 9.1). I used the 50-

megabyte concatenated sample sets (used for the dzip evaluation) since the experiments were meant to demonstrate the sustained throughput over different data types. Each experiment consisted of sending the 50-megabyte input across the network (using a 1Gbit/s limit so that throughput was not capped) and each test was repeated between 20-30 times. The result of the each test is the empirical Effective Output Rate (EOR) for the configured Scenario and Strategy combinations.

10.6.2 Data Type by Compressors

Figure 85 plots the EOR for four data types compressed with five different Methods as the thread count increases from 1 to 4. Clockwise from the upper left, Wikipedia, Facebook, Zero and YouTube data are compressed with each of null, LZO, zlib, bzip2, and lzma (xz) at the lowest strength level (to maximize throughput) using 32K chunk sizes. Confidence intervals were computed using the same technique described in Section 8.3.

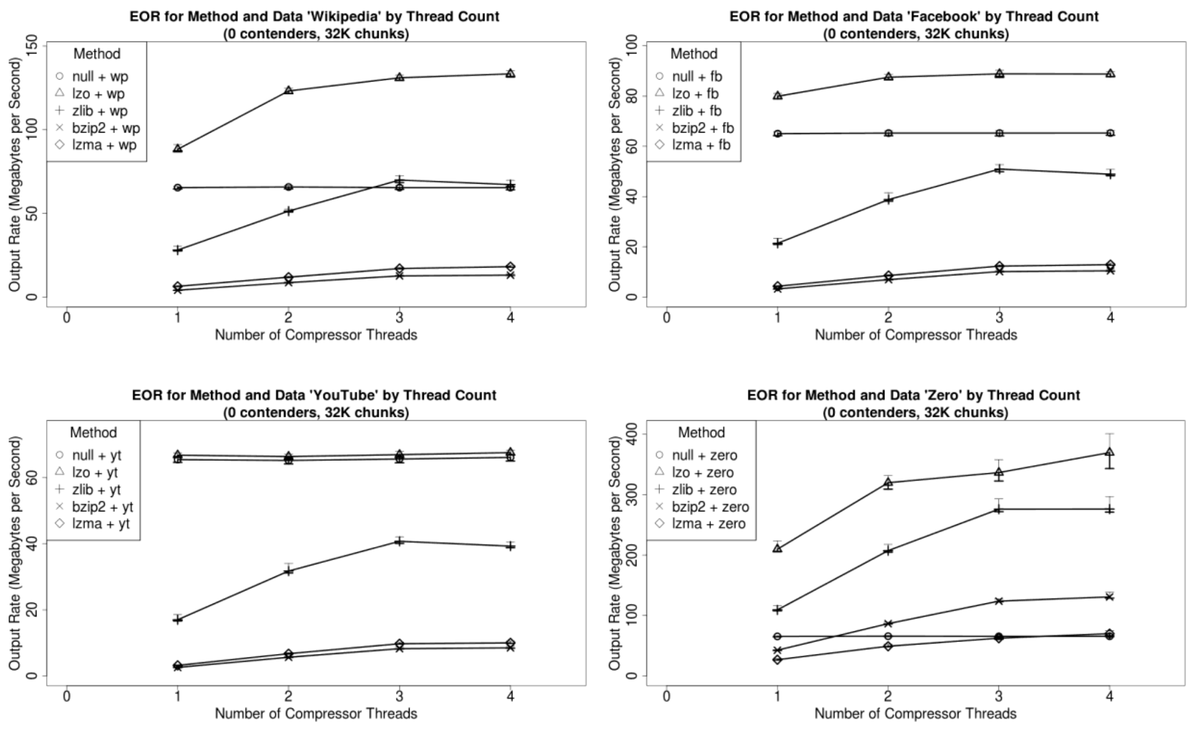


Figure 85. EOR vs. thread level of multiple Methods for one data type.

Null compression is stable at approximately 65MB/s regardless of the number of threads in use. Since null does not derive any benefit from compression, this shows that the maximum throughput of the current drcp/Datacomp implementation is about half that of gigabit speeds, which nominally have a maximum of 125MB/s. However, 65MB/s is still much faster than 12.5MB/s (the throughput of 100-megabit networks), so 100Mbit/s and 1Gbit/s are still able to serve as contrasting environments. Code optimizations and design choices such as using larger chunk sizes would almost certainly improve the maximum throughput of the implementation.

Since each thread performs roughly the same task in terms of data type and length, the best-case scenario for parallel compression is direct multiplicative improvement for each new thread. However, the overheads of parallelism, such as synchronization, means that there is some maximum number of threads that will improve performance for the given hardware and software.

For the drcp results discussed in this section, the number of useful threads is typically three, although some compressors improve with four threads in certain scenarios.

For Wikipedia data (upper left), LZO is far and away the fastest Method. LZO's worst performance is 88MB/s for one thread, which increases to nearly gigabit speeds with two threads. Above two threads, LZO's EOR (which includes the effect of compression) is greater than gigabit speeds, reaching 133MB/s with four threads (an increase of 51% over one thread). With three threads, zlib's performance grows from 28MB/s to 67MB/s (an increase of 139%), becoming statistically indistinguishable from null's rate. At the bottom are lzma and bzip2. Lzma's throughput ranges from 6.4MB/s with one thread to 18.2 with four (a 184% increase), while bzip2's performance ranges from 4.1 to 13.1MB/s (an increase of about 220%). However, even though the improvements for zlib, bzip2, and lzma were proportionally much greater than those gained by lzop, they did not change the best strategy.

For Facebook data (upper right), the same general trends are apparent, although the improvements are greatly diminished, because Facebook data is not as compressible as Wikipedia data. Lzop's performance shifts from 80MB/s with one thread to 89MB/s with two threads, an increase of only about 11%. With three threads, zlib moves from 21MB/s to 49MB/s, for an increase of approximately 128%. Bzip2 and lzma both start at lower throughputs, but increase about as much as they did while compressing Wikipedia data. The significant fact here is that the benefit of parallelism can depend on both the data type and the Method in question.

YouTube data is shown in the lower left plot of Figure 85. It clearly shows that the throughput of Methods *can be* improved by threading, even if compression is not possible. The dzip results showed that YouTube data does not compress well with the existing chunk sizes;

nevertheless, zlib, bzip2, and lzma all show EOR improvements with additional threads. Yet, lzop's performance does not improve significantly.

This is due in part to the fact that compressors impose a delay due to their computations. By threading compression, much of that overhead is "paid" simultaneously by each thread, enabling the network channel to be more fully utilized. Accordingly, faster Methods (which are less computationally bound) are likely to improve less with additional threads than stronger Methods that are more starved for CPU time.

The lower right plot shows the performance on "Zero data" as the number of threads increases. In this case, all Methods are able to match or beat null compression, although lzma is only able to do this with four threads. This shows that all compressors used in Datacomp are capable of matching the performance of null, if the data is sufficiently compressible. Interestingly, for this type of data, bzip2 is significantly superior to lzma, while for most other data types it is somewhat slower. Finally, it is noteworthy that for LZO and zlib, the confidence intervals widen with more threads. This is not surprising, and is likely due to an increase in variance because of the additional scheduling and locking overhead required for synchronizing the compression operations.

10.6.3 All Types by One Compressor

Figure 81 shows the EOR for four Methods (null, LZO, zlib, lzma) compressing all the data types (wp, fb, yt, ucww, ucwf, bin, mail, urandom[103]) as thread count increases from 1 to 4. All Methods use strength level "0" with a chunk size of 32K and no contender processes were running during the tests. It is important to take note of the different Y-axis scales.

[103] Zero data is not shown because the resulting EOR dwarfs the rest of the data types.

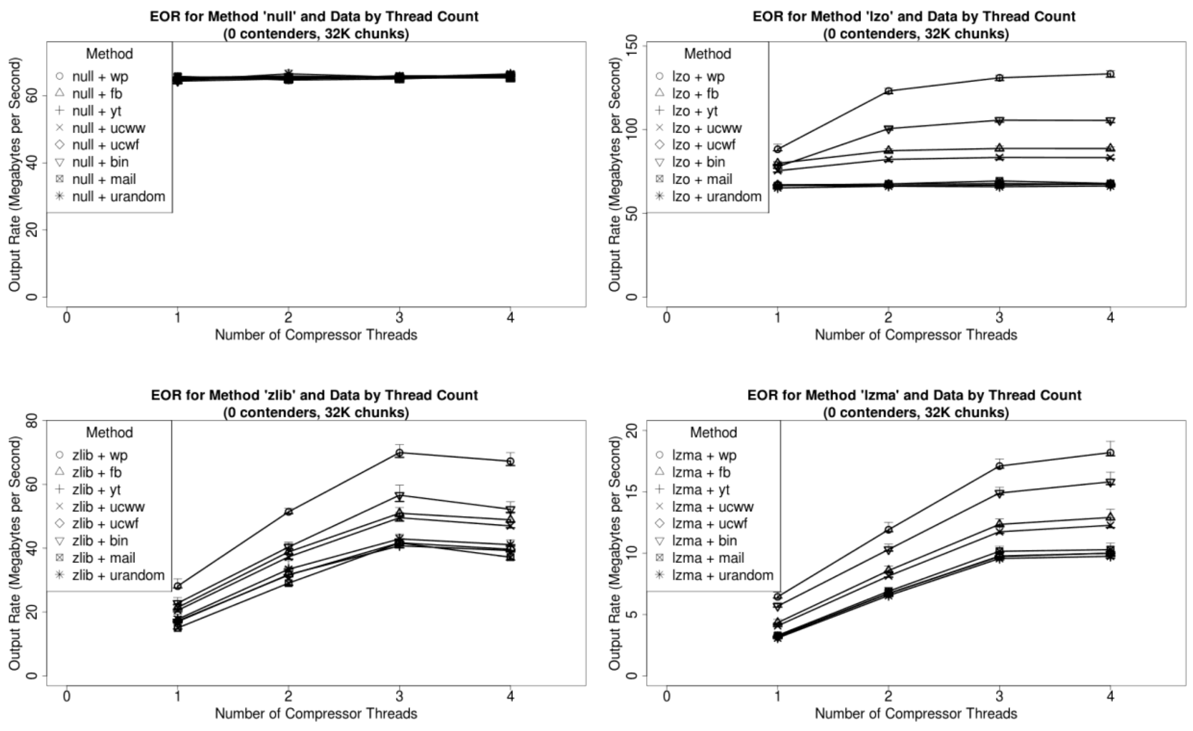


Figure 86. EOR vs. thread level – one Method for multiple data types.

There are several patterns visible in the data. First, performance for null on all data types is effectively the same, regardless of the number of threads. This is exactly as we would expect, since the primary value of parallelizing compression is to amortize compression overhead by performing multiple operations simultaneously. “Null compression” has no computational overhead, so parallelizing it has no real effect (when all other conditions are the same).

For other types of data, when an actual compression Method is being used, there is a clear trend: while performance improves and then tapers off, the actual performance results depend on the data type. Additionally, it seems that there is something of a “ranking” of types in terms of threading benefit that is mostly consistent across different compressors. For all Methods, the data type most benefitting from threading is Wikipedia, followed by binaries, Facebook, and User Web. The remaining types, mail, pseudo-random data, User Files, and YouTube, tend to be grouped closely together because they do not improve very much.

While there is a "ranking" in terms of threading benefit, the results are not the same across all Methods, nor are the effects of threading uniform between Methods. For example, the four "hard" data types (Pseudo-random, YouTube, User Files and Mail) are all virtually uncompressible by LZO. Accordingly, for those types, LZO has an output rate roughly equivalent to null compression, with little or no improvement with additional threads. However, the EORs for both zlib and lzma on those types improve dramatically with two and three threads (although they do not catch up with LZO). In the case of zlib, performance degrades slightly (but significantly, in the case of binaries) when increasing from three to four threads, while lzma performance continues to increase. Ultimately, the benefit of threading depends on (at least) the combination of the Method and the data type.

10.6.4 Chunk Size and Parallelism

Figure 87 shows one example of the effect that chunk size can have on EOR in conjunction with parallelism. Parallelism and chunk size together have a multiplicative effect on the amount of data being processed at one time, since Datacomp attempts to consume *(chunksize * num_threads)* bytes during each Opportunity. Figure 87 shows mail data compressed using 32K chunks (left) and 512K chunks (right).

With 32K chunks, zlib's maximum throughput is about 42MB/s with three threads. At four threads, performance falls to 37MB/s. Additionally, null and lzop are distinct, with lzop having a small but significant advantage with three threads (which degrades at four threads). With 512K input chunks, the most obvious effect is that zlib's performance increases to a maximum of 54MB/s and does not degrade with four threads. Slightly less obvious but nevertheless statistically significant, lzma's EOR is improved by 28%, while bzip2's is *reduced* by almost 13%. Thus, different chunk sizes *do* have a significant effect, and this effect interacts with parallelism in non-obvious ways.

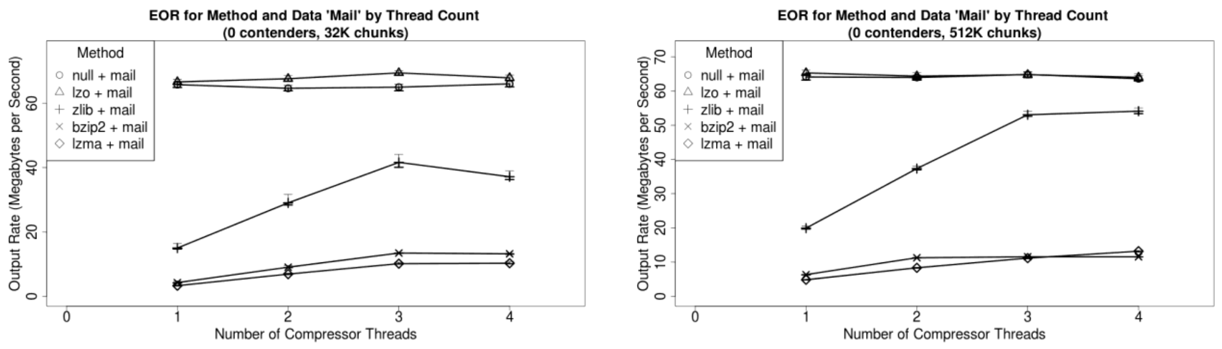


Figure 87. EOR vs. thread level and chunk size. Mail data and multiple Methods with 32K chunks (left) and 512K chunks (right).

10.6.5 CPU Contention and Parallelism

Figure 88 shows the effect that contention can have on Datacomp's performance in combination with threaded compression. It shows the use of drcp's Methods to compress and send binary data with no contention (left) and four bzip2 contender processes (right). While it may not be the most obvious element of the graph, the scale of the Y-axis is probably the most important piece of information, because it shows the reduction in throughput clearly. While null compression's output rate has until now been fixed around 60MB/s (as seen on the left), with four contender processes this rate is slashed to 10MB/s. Additionally, no Method is faster than null with one thread, although the strong compressors are meaningfully slower.

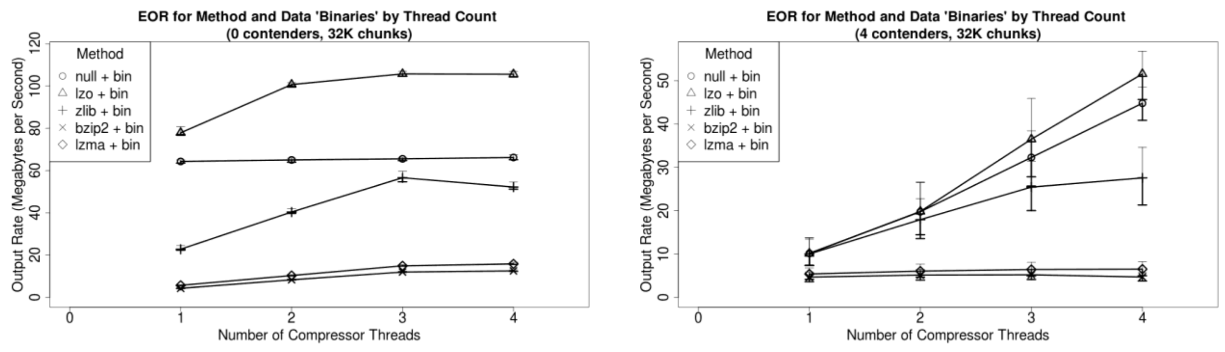


Figure 88. EOR vs. thread level – one Method, multiple types. No contention (left), Four contenders (right).

What is most striking, however, is that using multiple threads under contention improves throughput dramatically. Four threads improves null compression's throughput from approximately 10MB/s to above 40MB/s and lzop's to above 50MB/s (although the difference may or may not be statistically significant). Also apparent is that the variance increases with additional numbers of threads.

10.6.6 Discussion

This data serves to show that parallelism can improve the algorithm output rate (AOR) of many Methods, similar to increasing the CPU frequency (and interacting with many different factors). In fact, parallelism is really a means of providing more computational power to the same task in the same amount of time, which can result in saving time (or energy). However, more practically, parallelism seems like a potential path for improving the output rate of algorithms that are good compressors but are too slow to be competitive for a given environment.

This strategy can be effective, but it is not a panacea. Recall from our discussion of EOR that the available bandwidth (ABW), algorithm output rate (AOR), and the compression gain (1/CR) are combined in the following equation:

$$EEOR = min(ABW, AOR) * compression\_gain$$

One result of this is that Methods do not benefit from being *faster* than the ABW, but they are penalized for being *slower*. If Method *F* has a higher rate than the ABW but Method *S*'s rate is lower, threading *S* might be able to bring its output rate up to the ABW, ultimately enabling a higher EOR (if *S* has a better CR). However, threading will not improve *F*'s EOR because it is *already* faster than the ABW.

On the other hand, given two compressors that are well below the ABW (e.g., xz and zlib on a 1Gbit/s network), parallelism will improve them both, meaning that while parallelism will improve performance, it is not likely change the ultimate Method choice (because sending uncompressed will still be faster). Similarly, if one compressor is well below the ABW but another is well above it (e.g., lzop and xz on a 100Mbit/s network), threading is not likely to improve the slower Method enough to make it competitive. How these relationships work out in practice depends on the Methods available, the parallelization mechanism and the bandwidths of the communication channels.

10.7 Summary

Datacomp is able to significantly increase the efficiency of communication over channels using purely local mechanisms that rely on a minimum of hard-coded knowledge. Datacomp monitors local conditions internally, without the support of any other hosts. In addition to monitoring environmental parameters, Datacomp differentiates between different data types using a lightweight compressibility estimation technique that is broadly able to group data with similar performance properties. Using these environmental monitors and the compressibility estimator, Datacomp makes compression choices in real time based on a persistent history mechanism that learns as different compression Methods are used. Over a broad range of real-world data types and environments, Datacomp consistently improves throughput, saving approximately as much time and energy as the best static Strategy for the given Opportunity.

While saving any amount of resources is good, we would like to know how close Datacomp comes to the ideal Strategy. Typically, the best compression Strategy and its benefits are unknown. However, by comparing the results from the Comptool analysis tool to the results of Datacomp, I was able to determine how close Datacomp's independent and self-taught history mechanism could come to achieving results similar to the best compression strategies.

In the end, Datacomp's performance improvement is typically within 1-2% of the theoretical improvement predicted by Comptool's exhaustive analysis and greedy search for bandwidths up to 100Mbit/s. For example, it can improve the efficiency of Facebook transmissions by approximately 35% at low bandwidths and by 25% at 100Mbit/s. It can improve the efficiency of Wikipedia data by 72% at 1Mbit/s and almost 50% at 100Mbit/s. In cases when no compression is the best choice, Datacomp typically correctly identifies this and disables compression; for example, on YouTube data its performance is typically equivalent to not using compression.

Datacomp is typically able to "pay its own way," making up for its adaptation costs by improving efficiency through compression, parallelism, or its pipelined architecture. That Datacomp is "profitable" means that a system like it should be able to improve the efficiency of independent mobile devices like laptops and smartphones when both parties support Adaptive Compression. This could result in a significant savings of energy and other resources without requiring any hardware changes.

While Datacomp is a success, the current prototype is largely unable to improve the efficiency of a transmission at gigabit rates. This is for two reasons. First, mistakes are extremely costly at high bandwidths because even tiny delays impact a large number of bytes. Second, the cost of adaptation is fixed in the current prototype. As a result, adaptation cost overtakes any benefit that Datacomp could provide – unless the data is extremely compressible or the system is under heavy load. However, simple practical solutions to these problems exist that could be used in a fielded version of Datacomp, and a number of potential research solutions to these issues will be explored in future work. Ultimately, Datacomp overwhelmingly meets its goals for its target platform of "typical computers" in the "general computing environment."

11 FUTURE WORK

Comptool and Datacomp suggest and serve as a foundation for a large number of potential future research projects.

11.1 Improved Compressibility Estimation

Compressibility estimation is a virtually unexplored research area. Apart from William Culhane's 2008 work [48] and Datacomp's bytecounting method (described in Section 6.4.4), I am unaware of any other literature discussing data analysis methods explicitly designed to estimate compressibility more efficiently than performing the compression itself. While these functions may have seemed like a solution without a problem in the past, AC provides a valuable use case for effective mechanisms. New algorithms are relatively simple to implement and test, and could be added to Datacomp and Comptool by changing approximately 10 lines of code. Optimizing existing mechanisms would also be useful, as would investigating optimal bytecount sampling and aggregation methods. Finally, while existing methods output a single statistic, estimators that provide multiple values, such as a CR *and* an estimate of error or compression runtime, could be very valuable for prediction mechanisms and AC Models.

11.2 Enhanced Decision Models

Datacomp's current history database is a simple lookup table with rolling averages, and the quantization mechanisms are primarily divided equally across their range. These features could be improved and be made more dynamic, reducing the amount of human "expertise" included in the system. Basic extensions to the history mechanism could include tracking a confidence statistic along with average throughput, using a history function other than the rolling average (e.g., to improve robustness), improving upon the mechanism for avoiding local minima, and so on. Another enhancement would be for the history mechanism to track the throughput at which it was no longer able to improve efficiency, disabling compression at that level. This would solve the "gigabit problem" faced by the current implementation without requiring a fixed threshold.

Extending the "dynamic threshold" idea, the time decider would be more robust if the history mechanism managed its own quantization thresholds, such as the hypothetical bandwidth "compression cutoff" just mentioned or the existing bytecount compression cutoff (currently $b > 100$). In the simplest form, the Model would adjust the thresholds of a fixed number of quantization levels to maximize predictive power. However, it would be even better if the Model could add or change buckets according to observed performance over time. The database could start with a single "opportunity table," duplicating it when the Model determined that an environmental parameter or performance value was diverging in a significant way. This would not only tune the values in the database but the structure of the database itself to the observed performance caused by the data, environments, and underlying hardware of the given Datacomp instance. It could also shorten training time, since newly minted opportunity tables would start with valid data. Finally, it is also possible that completely different "history mechanisms" could be implemented using machine learning or other techniques, rather than the lookup table used in the current design.

11.3 Energy-flexible Hardware and Mobile Devices

Some future directions for Comptool and Datacomp center around evaluations on different hardware. In Section 9.2, I discussed how the best time and energy strategies obtained by Comptool were too similar to justify a dynamic energy strategy Model. However, results like this are tied strongly to the hardware used. Modern hardware is increasingly growing more flexible; for example, some newer Intel CPUs support distinct frequencies on a per-core level. If

LEAP instrumentation existed on these newer systems, Comptool could investigate the energy benefits of these features for AC.

For example, it could be interesting to compare the energy cost of a single threaded, fast compressor such as LZO with a multi-threaded, slower algorithm, such as zlib. By using a single-thread of LZO, we could target the energy-saving features of the hardware, allowing it to “sleep” the unused cores. This might result in lower CPU energy cost, even if it took more “wallclock” time. At the very least, these approaches would provide insight into the energy properties of newer hardware as they relate to AC workloads. However, if these techniques provided enough savings, they might justify the use of online LEAP for Datacomp.

Similarly, low-power, wireless mobile devices such as smartphones have a lot to gain from AC. Companies such as Nokia and Opera provide proxy-based AC, but this has security implications and may not actually reduce total energy consumption due to the overhead of supporting the proxy machines. It is not clear how directly Datacomp’s PC-based results will transfer to smartphones due to their different hardware, energy, and performance characteristics. Smartphones have limited batteries and rely on slower, energy-expensive wireless links for networking (both Wi-Fi and cellular), which increase the potential value and likelihood of efficiency improvements. However, they also have considerably less computational power than PCs, limiting the number of successful strategies and the number of environments in which they can be successful. Running tests in these environments would be a valuable exploration of AC for these devices.

Finally, hardware acceleration has been highly successful for certain tasks, such as graphics and cryptography. Hardware accelerators for DEFLATE compression exist, and it would be interesting to integrate Datacomp with hardware acceleration as provided by an expansion card or FPGA hardware. If hardware support could decrease the cost and improve the performance of compression, it could make AC even more compelling for future systems.

11.4 Testing and evaluation including decompression and negotiation

The experiments performed and described in this work focus on sender-side costs. However, in the real world, receiver costs are just as important, especially for low-power mobile devices. The ideal strategies for the sender and receiver may not be the same, and can change depending on the real-world relationships between the machines. For example, a customer of a website might expect the server to spend effort compressing data for the customer, while two smartphones communicating with each other might see each other more as peers.

Currently the sender in Datacomp makes all choices about what Methods to use. The receiver is primarily limited to choosing how much data to decompress at once. Instead, the sender and receiver could collaborate on the compression strategy based on their own goals and capabilities. This could still be transparent to the user, but would require some kind of negotiation process and communication channel. This channel could also be used to cooperate for other purposes, such as measuring the available bandwidth between the two hosts.

11.5 Improving POSIX Compatibility

While Datacomp provides enough compatibility with the POSIX `read()`, `write()`, `send()`, and `recv()` calls to support dzip and drcp, it lacks support for other system calls essential for most applications, such as `seek()`. As discussed in Section 6, adding `seek()` support to Datacomp should not be especially difficult. However, making Datacomp a truly "drop in" replacement library for legacy applications means supporting a large number of other behaviors (such as non-blocking IO) or other system calls, such as `select()`, which could prove difficult. Other functionality, such as memory mapping files (`mmap()`), shared memory or non-blocking IO may be impossible to emulate using a library in a 100% compatible fashion.

I plan to work through these issues as part of taking one or more existing pieces of open-source software, such as a backup tool or a VPN, expanding Datacomp's functionality to support the behavior of the original application. This project would be useful for AC research beyond Datacomp, as it would explore the interaction between existing API semantics and adaptive compression. For example, properly (or efficiently) supporting all required functionality might require changes to the Datacomp Stream Format (described in Section 6.3.2).[104]

11.6 Kernel Datacomp

Solving the problems inherent in improved POSIX compatibility would facilitate the next and most significant piece of future work: integrating Datacomp into the operating system. An AC-aware OS has a number of potentially significant benefits. By being integrated into the OS, Datacomp would be able to benefit user applications without requiring them to be modified in any way; AC would be truly transparent. Kernel Datacomp would also be more robust than the current design. Due to Datacomp's nature as an application library, it is technically just another part of the user application and so is subject to memory corruption and the effects of various bugs.

Kernel Datacomp would also be more efficient than a library because it would have direct access to available system information about throughputs, CPU load, scheduling and more, which is difficult and expensive (or impossible) for user applications to acquire. Kernel Datacomp might be able to share monitor and Model information between users and processes

[104] AdOC, by Jeannot, et al., [65] supported a broader range of these API semantics, so I would start this process by examining their approach.

while minimizing any attendant security issues. Placement in the kernel would almost certainly result in better support of the POSIX interface.

Multiple system components could benefit from an AC-aware OS. For example, an AC-aware operating system could compress data destined for the disk, choosing the compression strategy to save space or to improve throughput between the disk and memory. Compressed data, whether read from disk or downloaded from a Datacomp-using host, could be sent directly to the disk, without requiring decompression. Sometimes, a single compression operation would benefit the system many times over. For example, data that is read often but written rarely, such as executable binaries, could benefit significantly from compression because the compression investment is a one-time cost. When installing AC-aware binaries, the local machine might be able to benefit without even paying for the original compression.

Compressed reads from disk could left compressed in memory, page protected, until read by a process. Accessing the page would cause a soft interrupt, whereupon the kernel would transparently decompress the data for the process. This essentially describes a kind of compressed page cache (the "free" memory used by the operating system to store pages from disk), which could lead to a more efficient use of memory by virtue of an effectively larger cache. In turn, this could reduce the number of required disk reads, enabling the system to save more energy.[105]

For writes, data could be compressed behind the scenes as part of the write process. This need not delay performance, since the compression could be performed by the kernel in memory prior to the data being flushed to disk. Alternatively, raw, uncompressed data could be written directly to disk when necessary, with a "housekeeping" process compressing data when spare

[105] A previous project of mine, Cryptkeeper [114], showed that the kernel could do this kind of work with cryptography on a large scale at a surprisingly small cost.

CPU cycles were available. This housekeeper could also be used to transcode data from one compression Method to another, such as from a format originally used to save time, to one used to save space. As long as these tasks were performed using cycles that would have gone to waste (or would not be noticed, such as when charging), the entire process could improve performance overall.

Building a Kernel Datacomp mechanism would be a huge undertaking. However, it could potentially be done incrementally, such as by starting with an AC file system or AC networking stack, similarly to how our lab built the Data Tethers OS [116]. Given the improvements achievable with the current, user-space Datacomp and the benefits of being integrated with the OS, Kernel Datacomp could be an extremely valuable mechanism for improving efficiency.

11.7 Datacomp Lite

Finally – and at the opposite end of the spectrum from these other future work concepts – is the concept of a highly simplified version of Datacomp. Datacomp's current design, from the quantization sizes to the number of Methods available, is focused on being comprehensive, dynamic, and not relying on hard-coded advice. However, from a practical perspective, a little hard-coded expertise might go a long way.

Datacomp's current design results in a large system with a long training time and other drawbacks as detailed in this work. An interesting project would be to distill Datacomp to its smallest possible footprint that is still capable of performing as well as the existing version on various types of hardware. For example, Datacomp Lite might only support null compression, LZO, and zlib, or it might consider two levels of CPU load: unloaded and fully loaded. Its history mechanism might keep an ordered list of choices rather than rolling averages, and so on.

Regardless, the process of creating Datacomp Lite would be instructive in a different way than building the original Datacomp because it would focus on making the design choices with the highest value, emphasizing the factors that are truly essential for effective AC.

12 Conclusion

The primary goal of this project was to create an Adaptive Compression system capable of using compression to improve the efficiency of communication operations for users of real-world, general-purpose computers and typical data – the kinds of devices and information you, your friends, and your grandmother use on a day-to-day basis. While previous works have shown that AC can be a practical means for improving efficiency in certain circumstances, they have typically focused on specialized environments, limited data types, and have often required dependencies or assumptions that precluded or reduced their usefulness in the "real world."

To realize my goal I created and evaluated Datacomp, a successful proof-of-concept AC system that improves efficiency significantly for commodity computers in typical environments without relying on *any* external support, additional hardware or precomputed internal models. Datacomp accomplishes this by monitoring relevant environmental conditions, making compression choices, and carrying out compression operations itself. Using its internal monitors and compression choices, Datacomp can optimize for space by identifying the best greedy compression strategy based on its available Methods. More significantly, Datacomp can optimize for time and energy by making choices based on input from its monitors and a fast, persistent historical performance Model that it builds based on the results of local compression operations.

Datacomp is designed as a user-space library that can be integrated into real-world software. The library wraps the function calls `read()`, `write()`, `send()` and `recv()`, transparently performing AC on the payloads to optimize them for a specified resource (e.g., space or time and energy). Because Datacomp respects the legacy APIs for these system calls, integrating Datacomp into software requires very few changes.

I created two example applications based on Datacomp to demonstrate its power, flexibility, and simplicity. Dzip is a file based compression utility typically used to adaptively maximize compression, while drcp is a TCP-socket based transmission utility typically used to maximize throughput. However, both utilities are completely goal-agnostic. In other words, dzip can be used to maximize write throughput and drcp can be used to minimize size simply by specifying the desired goal to Datacomp. Furthermore, while I built these applications as simple demonstrations, Datacomp is intended to be useful for any type of application wanting to improve the efficiency of file or networking I/O, such as a VPN, productivity tool or server-client architecture. By evaluating the performance of my test applications on realistic workloads, I could quantify the performance of Datacomp.

A secondary goal of mine was for Datacomp to be a platform for AC research going forward. To that end, I studied the existing AC literature in order to identify the components and tasks common to all AC systems. This provided an opportunity to create terminology to describe common components of the AC problem and solution spaces.

I identified three primary elements of the AC problem space: Data, Methods and Environments. "Data" forms the input to an AC problem; it is the data that may or may not be compressed and it can be identified by any properties of the input that are relevant to compression performance. "Methods" are the distinct compression mechanisms that could be used to compress the Data, including "null compression." "Environments" are composed of all the software and hardware properties that affect the behavior of the compression Methods.

These components can be combined in order to better understand, model, and analyze the AC problem space. Data and Environment combine to form an "Opportunity" – a unique juncture at which some compression Method may be applied with varying outcomes. A Method applied to an Opportunity results in a discrete “Event,” which itself is composed of a number of “Results” – every performance property or statistic that might be useful for decision making or otherwise impact effectiveness. A sequence of Opportunities forms a “Scenario,” such as transferring a file across a 1Mbit/s network. Similarly, a sequence of Methods with respect to some Scenario is a “Strategy.”

Similarly, I identified four high-level components of AC solution spaces. “Mechanisms” comprise the functional skeleton of an AC system and its software interface, including its API and protocols it uses for communication. “Monitors” are the entities responsible for providing the AC system with information for making decisions. “Methods” are (again) the distinct compression mechanisms available to the AC system. Finally, “Models” are the components that take information from Monitors, choose a Method, and task the Mechanism to perform the operation.

Identifying these common components allowed me to compare prior work with a unified perspective, illuminating similarities and differences between past systems. It also allowed me to build Datacomp in a more modular fashion. This enables the exploration of new AC designs by modifying or replacing one or more of Datacomp’s core components, rather than having to create a new system from scratch.

While adding new Monitors is straightforward, Datacomp already includes Monitors to obtain the CPU load, frequency and available bandwidth. Additionally, Datacomp uses a new compressibility estimator called the “bytecounter” which can estimate the compression performance of various input types. Importantly, using the bytecounter is significantly less expensive than performing compression. Together, these monitored values serve as input to Datacomp’s decision Models.

Datacomp currently includes three decision Models. The Manual Model is primarily used for testing and development. It non-adaptively applies the specific compression mechanism requested by the user. The Space Model attempts to optimize for space over time by performing a greedy brute-force evaluation of all compression options, keeping the best results. The Time Model (which also serves to optimize for energy) uses the combined input from every Monitor as a key to a persistent history database of past compression results. The Time Model selects the Method with the best average result, performs it, and updates the history mechanism with the results.

To explore the theoretical value of AC for my target environment and to validate the results of Datacomp, I also created Comptool, a standalone compressibility analysis tool that performs compression on an input and uses a greedy mechanism to identify the best compression strategy for a specific goal. Comptool includes LEAP energy measurement capabilities, allowing Comptool to accurately measure the energy effects of AC choices in addition to time and space costs. In addition to being useful for AC, Comptool should be useful for developers or engineers seeking to optimize the performance of their non-adaptive systems.

In order to test Datacomp and Comptool in as comprehensive and compelling a way as possible, I created workloads that represent a broad range of data types and execution environments. I collected a hundreds of megabytes of nine different data types. Five types represent well-known, extremely common types of data. These include binaries taken from Ubuntu Linux, a large amount of personal email including attachments and web data collected from Wikipedia, Facebook and YouTube using a controlled method. Two types represent data that is unarguably realistic in that I collected it from six volunteers. These types included live web surfing data and random personal files. Finally, to include data with best- and worst-case compressibility properties, I collected two artificial classes of data: “pseudo-random” and “zero,” drawn from `/dev/urandom` and `/dev/zero`, respectively.

To perform experiments in a broad range of real-world environments, I sent samples of different data types through Comptool and Datacomp while modulating the experimental environment between tests. Five bandwidth levels were chosen to represent the common types of networks in use today: 500Kbit/s and 1Mbit/s emulated cellular data rates, 6Mbit/s emulated home DSL or cable Internet, 100Mbit/s matched the common legacy LAN rates, and finally 1Gbit/s served as the highest rate. While “Wi-Fi” bandwidth is not explicitly represented in this list, it usually falls between 1 and 100Mbit/s, depending on the technology, signal strength and so on. CPU contention was artificially added to the system by running parallel processes that consumed CPU cycles during testing. Finally, I tested the effect of CPU frequency by locking the CPU to a static frequency during tests of different types. On a different system, I tested dynamic frequency control by allowing the kernel to control the CPU frequency during the tests.

Datacomp and Comptool currently support five distinct compression families: “null compression” (which simply copies the data), LZO (an extremely fast but comparatively poor compressor), zlib (a widely-used DEFLATE implementation and the underlying library for gzip), bzip2 (a strong compressor using the Burrows-Wheeler transform), and lzma (a computationally costly but highly effective algorithm underlying the xz utility). Each compressor is configured to support three different strength levels: “fast,” “medium,” and “strong.” Datacomp also supports parallelizing any compression operation using between two and four threads, regardless of whether the library itself supports parallelism.

Under the hood, both Comptool and Datacomp function by quantizing data – cutting the input into smaller chunks and performing compression on those chunks before sending it.

Datacomp and Comptool support the same “chunk sizes” so that their results are comparable: 32K, 64K, 128K, 256K, and 512K. Data less than 32K but greater than 512 bytes is still compressed (unlike some AC systems), while larger input (e.g., 1M) is consumed using multiple threads or subsequent operations. Comptool executes every possible Method (i.e., combination of compression family, chunk size, strength level, and thread count) on each chunk, only finding the best strategy after the fact through offline search. In contrast, Datacomp uses its Model and Monitors to choose what it believes to be the best Method in real time.

In other words, Comptool and Datacomp perform the same tasks, but acquire their results in very different ways. Comptool doesn’t improve efficiency, but it does find the greedy best solution for an experiment. On the other hand, Datacomp actually performs adaptive compression, but cannot speak to the optimality of its results. By comparing the two, we can discover the best strategies with Comptool and use those results to evaluate the real-world performance achieved by Datacomp.

As far as I am aware, this collection of compression Methods, experimental Environments, and workload Data is far larger and more diverse than any previous AC system or evaluation, particularly in terms of targeting real-world systems and workloads. Similarly, I believe that the use of an empirical solution (Comptool) as an evaluation tool for an AC system is novel. The range of chunk sizes and compressor-agnostic parallelism provides a much wider range of Method choices for both Comptool and Datacomp than in past work. The broad range of bandwidths, CPU contention levels and frequencies used resulted in testing of AC mechanisms in a more diverse and comprehensive range of Environments than many other systems. The nine different Data types, sampled and assembled in different ways, run the gamut of compressibility from trivially easy to pathologically difficult and contain enough data for many unique experiments. Finally, the integration of LEAP energy measurement into Comptool provided a level of energy use detail unseen in previous work.

Within these experimental Environments, the Comptool tests demonstrated that AC can indeed provide significant gains for many Data types, including some types that are not obviously compressible. In cases where data is truly uncompressible, Comptool properly identified null compression as the best Method, meaning that Comptool's "best" strategy would never result in a loss of efficiency due to pointless compression. Furthermore, Comptool's tests established that the best gains for compressible Data across Environment changes can only be achieved by changing the compression Strategies and in some cases by combining multiple compression Methods over the course of a Scenario.

For example, Comptool estimates that at 1 and 6Mbit/s, compression can save approximately 34% on Facebook data, primarily by using gzip compression. Since gzip is already the standard for web traffic, this is not revolutionary. However, at 100Mbit/s, Comptool shows that gzip compression is statistically equivalent to not compressing at all, but that using LZO can still improve efficiency by 27% over null compression. Even at 1Gbit/s, when gzip imposes a 300% penalty to throughput, Comptool demonstrates that performance could still be improved by as much as 14.3% using a mixture of LZO and null compression. Other Data types have even more impressive results, such as a 74% improvement for Wikipedia at 1Mbit/s and a 28% improvement at 1Gbit/s.

LEAP technology facilitated a comparison of the best energy and best time Strategies. To do this, I performed separate experiments using Comptool to optimize for time and energy. My results showed that, within the experimental constraints, the energy benefit of the energy Strategy versus the time Strategy was less than the cost of performing the energy analysis. This informed Datacomp's design in that it uses the same decision mechanism for both time and energy, allowing Datacomp to save energy without requiring LEAP hardware or hard-coded energy Models.

In general, Comptool showed that compressible Data benefits from stronger compressors at lower bandwidths, faster compressors at higher bandwidths, and a mixture of no compression and fast compressors at the highest bandwidth. Perhaps the most significant result from Comptool is that most compressible Data can theoretically benefit from compression at gigabit rates if the right choices are made. Comptool also showed that the value of specific choices, such as which chunk sizes or algorithms to use for various tasks, varied across Data types and Environments. In other words, these results show that using static compression Methods over varying Opportunities results in both missed occasions for improvement and losses that could have been avoided.

Comptool also shows that some Methods are significantly worse at compressing certain types of Data than other Methods, regardless of the Environment. Performance may be similar for certain sets of Methods and Data, such as LZO and gzip when processing Facebook and Wikipedia data. However, performance is decidedly *not* similar for other types, such as LZO and gzip processing binaries and mail. This shows that linear models relating compressor performance – without considering data type – can result in a loss of achievable efficiency.

While Comptool's ability to find successful compression strategies is impressive, it is not a real AC system. First and foremost, Comptool's results do not incorporate the cost of determining the best strategy. They are derived by using an offline greedy mechanism to find the best results in an exhaustive set of discrete compression results. This "best-case" result is by design, so that Comptool's analysis is not tied to any particular decision mechanism. This allows researchers to see what is possible without being tied to a particular AC mechanism.

When we think about what is possible with AC, we can identify two hypothetical goals. The "core" goal is that AC always performs as well as the best static strategy, changing the strategy as necessary to maximize efficiency. This would improve overall efficiency, assuming that the best strategy changes. We can also identify a "bonus" goal, where AC *outperforms* the single best static strategy for a given environment by virtue of quantizing the compression operations.

While Comptool meets its core goal of always achieving performance *close or equivalent to the single best static strategy*, it only sometimes identifies a strategy that is statistically *better* than the best static strategy. Because Comptool determines its results with comprehensive analysis, this suggests that the "core goal" – matching the Results of the best Method for an Opportunity – may be the best result we can hope for when using the type of quantization used by Comptool, Datacomp, ACE, and other AC systems.

I used Comptool to evaluate Datacomp's performance in two phases. First, I tested Datacomp's ability to maximize for space with dzip and the Space Model, which essentially performs a dynamic version of Comptool's brute force search in order to maximize compression. Dzip's results match those obtained by Comptool, however they both showed that quantized AC as used by Comptool and Datacomp does not result in improved space savings with the given chunk sizes and greedy heuristic. Instead, the results identified the 'xz' compressor as almost always achieving the best compression. Because dzip's "brute force" search mechanism took many times longer to achieve the same result, it seems that quantized space optimization is not a profitable strategy (at least with a greedy search mechanism), but that systems concerned with optimizing for space should simply use xz or a different strong Method.

The dzip evaluation also uncovered an interesting result; "strong" compressors like bzip2 and xz achieve compression ratios that are generally statistically similar to "weaker" compression Methods like zlib with the chunk sizes configured for these experiments. However, they benefit *significantly* from input sizes larger than 512K (Datacomp and Comptool's largest quantization level). By submitting even larger inputs (on the order of megabytes), xz was able to improve compression of YouTube data up to 47% over 512K chunks. In other words, Comptool and dzip demonstrate that very large quantization sizes could open up unique performance options for AC systems, even though they also come with risks.

While dzip uses Datacomp to optimize for space, drcp primarily uses Datacomp's Time Model to improve the throughput of communication operations. My experiments showed that for the bandwidths 1Mbit/s, 6Mbit/s, and 100Mbit/s, drcp is *consistently able to improve efficiency to within approximately 1-3% of the "ideal" improvement as determined by Comptool*. For example, at 1Mbit/s, where Comptool shows that Wikipedia data can be improved by about 74%, drcp saves around 72% in actual use transmitting the data across a physical network. At 100Mbit/s, Comptool predicts savings of 52%, while drcp achieves an improvement of 49%. When data is not compressible, drcp's results are almost always statistically similar to null compression for these bandwidths.

While drcp's performance is close to the best known strategy for most bandwidths, the current Datacomp infrastructure and test hardware is unable to keep up with the throughput at 1Gbit/s, unless there is heavy CPU contention (which reduces throughputs in general). At unloaded gigabit speeds, drcp wastes between 1 and 10% because the cost of adaptation is too large relative to the low cost of transmitting uncompressed data at 1Gbit/s. As a result, even though Datacomp usually makes the "right" choice for drcp, it still costs too much. Additionally, while suboptimal choices can still be profitable at lower bandwidths, prediction errors at gigabit speeds are difficult to hide because there is so little room for error.

One potential strategy for overcoming this issue is to spend less time analyzing data at higher throughputs (such as by "bytecounting" fewer bytes), only applying the fastest compressor if a streamlined sampling process suggests that the data is highly compressible. A more heavy handed "solution" would be to instruct drcp to simply not perform compression or data analysis if the time cost of adaptation is too high for the current bandwidth. While these strategies would be practical solutions for a fielded Datacomp system, it represents the kind of "hard-coded choice" that I chose to avoid for this project.

Ultimately, Comptool shows that AC is necessary in order maximize efficiency for throughput, because the best compression Method for a scenario changes based on many factors. It also shows that static Strategies are not ideal, because performance can differ from system to system and is highly Data- and Environment-dependent. Finally, Comptool shows that improvements due to AC can be quite significant.

While Datacomp has room for improvement, it nevertheless achieves significant improvements at the most common bandwidths for typical users. Datacomp conclusively demonstrates that AC can indeed improve efficiency for commodity hardware and real-world workloads – without having to rely on external hosts or special infrastructure that does not exist in the real world. Furthermore, this improvement is not some incremental boost of a few percent; for bandwidths between 1Mbit/s and 100Mbit/s, it is able to improve performance up to 74% over no compression and is typically within 1-3% of the best known solution. Finally, both

Comptool's analytic abilities (including LEAP instrumentation) and Datacomp's efficient and modular design can serve as a guide and a platform for AC research going forward.

Gains like these from AC are literally *waiting* to be reclaimed from our computer systems. Given the scope of this improvement and the fact that Datacomp was able to reclaim it without any external support, designers and developers should be thinking about Adaptive Compression not as a neat trick, a hack or a curiosity, but as a legitimate systems technique for improving efficiency. A widely available library like Datacomp would enable the use of AC in virtually any application, without requiring OS-level integration. This would allow incremental deployment and help familiarize developers with AC issues. However, an Adaptive Compression system like Datacomp integrated into the operating system would be able to transparently improve efficiency in many new ways, such as between storage and memory, storage and network, and more. A "Kernel Datacomp" would also be more robust and secure than AC as a user library.

In conclusion, Comptool and Datacomp show that Adaptive Compression is a practical and powerful technique for improving efficiency. Furthermore, they show that AC can be effective outside of the lab, in the real world, with commodity hardware. Continuing to design common applications and protocols to use compression in an unsophisticated, all-or-nothing fashion or expecting users to manage compression manually is simply *not the most efficient approach*. This work shows that, as we continually look for ways to improve the space, time and energy efficiency of our computers, devices and networks, we should be considering Adaptive Compression.

APPENDIX A: DATACOMP INTERNALS

This dissertation touches on many different topics. One topic to which it doesn't speak is *software engineering*. I haven't delved into the internals of either Comptool or Datacomp because the software itself is not the primary contribution of this work and because I make no claims about its optimality. However, it may be illustrative for people thinking about this work (or designing a new AC system) to have a little more detail about Datacomp's internals. To that end, this Appendix discusses the modularity and interface between Datacomp's internal components. At the end of this appendix, I provide resources for people interested in working with Datacomp in the future, which will ultimately include source code and more comprehensive documentation.

13.1 Datacomp Component Interfaces

Datacomp is a user space library, which means that Datacomp's components are a part of the application itself. This design choice was made for simplicity, as opposed to a kernel implementation, and for maximum throughput, as opposed to a daemon with some kind Mechanism requiring Inter-process Communication (IPC). However, it also means that the modularity or abstraction boundaries between components are "soft." In other words, abstraction boundaries within an application can be ignored, unlike the "hard" boundaries that exist between applications and the kernel. (For example, this is why libraries can crash your applications.) Additionally, Datacomp is a prototype developed first to complete my research and only secondarily as a platform for future work. As a result, sometimes the separation between components is not only "soft" in a software engineering sense, but is more conceptual than actual in the current implementation.

In the following sections I will describe the abstraction and interfaces for the Methods, Models, and Monitors. I do this to provide insight into how Datacomp as an AC system is separated into components and how Datacomp might be able to be extended. At the same time, I try to highlight differences between the current state of the prototype and the future direction I see for Datacomp's development.

13.1.1 Methods

Datacomp supports arbitrary compression libraries via wrappers that translate between the API provided by each compression library and a simple, uniform Datacomp Compressor API. The reason these wrappers are required is that most compressors generally have *similar* but not *identical* APIs. To compress, the application typically provides input and output buffers in addition to a length (and perhaps some options). Decompression usually requires fewer options since the decompressor can identify when the input has been fully decompressed.

While similar, the interfaces are typically unique enough that they cannot be used in place of one another. For example, zlib's primary argument is a pointer to a structure that has been initialized and set to contain or point to the necessary data and variables:

```
deflate(&strm, Z_FINISH)
```

`Z_FINISH` is a flag that tells the compressor to "flush" the operation – to complete it and emit any pending data. In contrast, LZO accepts the parameters individually, rather than as part of a structure:

```
lzo1x_1_compress(in, in_len, out, &out_len, wrkmem)
```

This LZO function does not require a flag like `Z_FINISH`.

Additionally, the initialization and cleanup steps for compressors differ (allocating and de-allocating memory, setting and resetting variables, etc.). A naïve approach would be to write custom code for each Method which is then "baked in" to the Datacomp code responsible for calling compressors. However, this hard codes Method choices into Datacomp and complicates its Mechanism code (since it must always be selecting the appropriate code for compression operations). A more flexible approach is to use the aforementioned translation layer so that Datacomp need not itself contain custom code to handle each library.

Datacomp calls the selected compression or decompression Method for an Event using a pointer to a function with a specific interface. This function is responsible for allocating the appropriate resources, performing the compression operation (decoding the strength argument into valid parameters for the library) and de-allocating the resources after use. (It will eventually be responsible for handling error conditions in a consistent fashion, but currently aborts on any error.) In this way, any compressor that can be made to compress or decompress within one function (including any required setup and teardown) can be supported by Datacomp with a minimum of effort.[106]

Each discrete compression library supported by Datacomp (or supported in a unique mode by Datacomp[107]) requires a wrapper that provides uniquely named functions for the Method. For example, the simplest Method is for "null compression," provided by the "wrapper" `dc_null.c`. It provides the functions `null_compress(void *arg)` and `null_decompress(void *arg)`, where `arg` is a `void` pointer to the Datacomp-specific

[106] This currently results in some loss of efficiency. For example, if the wrappers or the lower-level libraries could allocate memory only once, freeing it at shutdown time, compressor overhead may be able to be reduced.

[107] For example, Datacomp uses zlib to provide zlib strengths 1, 6 and 9. Zlib also supports some lightweight modes (e.g., Z_HUFFMAN_ONLY and Z_RLE). Since the current Datacomp mechanism supports three "strengths" per compressor family, I would need to create an additional wrapper for zlib that used these modes instead of the more traditional strengths. Alternatively, Datacomp could be modified to support more strength levels per compressor family.

`comp_task` or `decomp_task` structures, as appropriate. This structure contains the necessary elements for compression (e.g., pointers to buffers, input lengths and strength settings[108]) but also includes some Datacomp-specific variables.

At startup, Datacomp initializes arrays two arrays attached to the DCFD, `comp` and `decomp`, with one slot for each supported compressor family (each Method group provided by a discrete wrapper). These arrays are indexed by the "code" for each compressor. For example, `comp[0]` is set to point to `null_compress()` and `decomp[0]` is set to point to `null_decompress().`

Datacomp also initializes structures to be used as inputs for the compression functions. Specifically, it creates two arrays; one of `comp_task` and one of `decomp_task` structures, which are the input types for the respective Method functions. The number of slots in each array is based on the constant `NTHREAD_SLOTS`, which determines how many compression and decompression threads may be running in a given Datacomp instance. When Datacomp wants to use a particular Method, it either chooses the Method via a Model (if compressing) or determines the Method from the Datacomp Stream Header (when decompressing), looks up the appropriate function in the array, sets the members of the structure (for each thread if appropriate), and calls the function with the `comp_task` or `decomp_task` structure as the parameter.

The reason for this odd design is twofold. First, it keeps all necessary parameters for a compression or decompression Method call within one structure, which makes it easy to organize and access the Methods and resources in arrays. Because the API Methods accept the same kind

[108] The chunk size is not an explicit option for the compressors, but is instead controlled at the Mechanism level (by varying the amount of data provided to the compressor). In this way, the "chunk size" of Datacomp is about input amounts, not the "block size" of the underlying algorithm (which may or may not be adjustable).

of structure (regardless of the underlying compressor), we can allocate these arrays once at startup and reuse them during the lifespan of the DCFD.

Second, having the API functions take a single pointer as an argument makes it convenient to call the Method functions as POSIX threads. Threads are started using the function `pthread_create()`. However, `pthread_create()` only accepts four arguments. The first two relate to the thread itself, leaving only two arguments for starting a function – a pointer to the function being started as a thread, and a pointer to all the arguments for the routine. Passing a pointer to a structure makes it easy for the function to "unpack" multiple parameters of various types from one pointer.

Since Datacomp compartmentalizes Methods and their data in this way, very little work must be done to launch compression Methods in their own threads, with synchronization happening as part of Datacomp's Mechanism. When a compression or decompression routine has finished executing, the resulting output can be found at the relevant pointers in the relevant `comp_task` and `decomp_task` structures. From there, they are collected by Datacomp's Mechanism and are marshaled for transmission or return to the calling application.

13.1.2 Models

Models are the main decision-making components of Datacomp. Like Methods, Models are called using a function pointer , `dcfd->decide`, which is pointed to a function during creation of the Datacomp File Descriptor (DCFD) by `make_dcfd()`. The function pointed to depends on the resource priority chosen when the DCFD is created. Rather than an array of Models, Datacomp currently supports only one Model at a time. In addition to the function pointer, the DCFD has a `void` pointer that is used as a scratch space for whichever Model is active during the lifespan of the DCFD (i.e., instance of Datacomp).

Each Model in Datacomp is contained in a separate source file and is called using a standard set of functions, including initializer and destructor routines. The initializer (e.g., `NAME_decider_init()` where “NAME” corresponds to the Model’s name) initializes the Model-specific scratch space. This scratch space is a “black box” from Datacomp’s perspective; it can be used by the Model for any purpose. For example, the space Model stores the Event with the best CR in memory allocated here. The destructor function (`NAME_decider_destroy()`) simply frees the resources allocated for the Model as appropriate and is called when the DCFD is de-allocated.

The primary function for a given Model is `NAME_decider()`. For example, `time_decider()` optimizes for time/energy. The decider function is called by Datacomp as a result of an application-level call to `dcwrite()`. Unlike Methods, Models are executed in the main Datacomp thread. The Model accepts a number of arguments, including the buffer, length and flags originally submitted when the application called `dcwrite()`. The Model arguments also include a pointer to the buffer (outside of the scratch space) where the output will eventually be written (determined by Datacomp). Finally, the Model receives a pointer to the DCFD itself, which is a direct (albeit heavy-handed) way of making sure that the Model can access any necessary resources, including Monitors, Methods, and so on.[109]

Each Model is only *required* to perform only two tasks: (1) choose and set a compression Method by setting the members of a special Method-defining structure and (2) launch compression using a special function, `thread_launcher()`. Launching compression ultimately calls the Method wrappers as POSIX threads as described in the previous section.

[109] Untangling the minimum components and capabilities required by Models in order to improve isolation and modularity – without degrading performance – is one of my goals prior to a public release of Datacomp.

After launching compression, the Model returns to `dcwrite()` the number of bytes of data sent for compression and transmission.

While the Model is not *required* to perform other tasks, there are no restrictions on what the Model can do in the service of choosing and launching the compressors. For example, bytecounting is called from within the Model since not all Models require it. This means that changes to Datacomp, such as disabling bytecounting and Method choices above a certain bandwidth, or "cooldown" strategies used to reduce the number of bytes analyzed would be contained entirely within Model code.

Another example of optional work the Model can perform includes consulting other Monitors, such as reading the CPU load or frequency or obtaining an available bandwidth estimate from the ABW monitor.[110] Similarly, the time Model consults the history database within its decider function. In addition to any Monitor provided by Datacomp, Models can perform other timing and inline monitoring as desired. For example, Models can time the bytecounting operation or, in the case of the space decider, may actually perform test compressions in the search for the best choice.

If desired, Models can handle compression internally rather than having Datacomp launch the compressors. In this case, Models perform compression (using the function pointers and structures as previously described) and call `thread_launcher()` along with a structure indicating the Method used and a special flag indicating that compression has already been performed. The space Model uses this mechanism to submit the output of the Event with the best CR directly to Datacomp so that the compression does not need to be re-executed.

[110] Code reading the CPU load and frequency and ABW is currently in `dcwrite()` *prior* to calling the Model. This is not for any practical reason but is simply due to evolution of the code base. It will be moved into the time Model in subsequent versions. In fact, having it outside the Model complicates Datacomp's code, since for accuracy's sake it must be skipped when using Models that do not require it (such as the manual Model).

Because the Model's decider function returns immediately after choosing a Method and launching compression, it can be necessary to use a callback function to complete the Model's adaptation task. For example, the time Model needs to update its history database following the completion of the compress-and-send operation it launched. However, obtaining the overall result requires waiting for the end of the pending send operation which is outside the scope of the decider function. Waiting for the result would be inefficient for Datacomp, so in the current Datacomp code, a function `dc_update_database()` is called after the completion of the send (if the time Model is enabled). This code is currently "inside" the Mechanism, protected with a conditional. Future versions will formally abstract this out so that Models can provide one or more callbacks for execution at different points in Datacomp's operation.

13.1.3 Monitors

Monitors are the least isolated or abstracted component of Datacomp. This is partly due to the organic evolution of Datacomp's code, but also because of the variation and generality of Monitors. The simplest monitors are just bits of inline code that measure and return a value. Many of these "Monitors" are built in to Datacomp because they require access to the Mechanism itself. For example, neither Models or Methods could measure the time required to perform a specific `send()` operation because that operation happens in the Mechanism. However, Model-specific Monitors (such as timing of a bytecounting function inside a decider function) could be included in specific Models wherever desired.

Increasing in complexity from these "one-liners" include Monitors that are called as discrete functions. The bytecounting function used by Datacomp is a perfect example of this, as it is completely separate from Datacomp or its components. The bytecounting code (itself in its own source files) is used by both the time and space Models (in the latter for debugging and research rather than as a necessity) by including the source file and calling it as a function. This is a straightforward way to share these kinds of Monitors between Models.

Finally, the main Monitors included in Datacomp -- the CPU load and frequency Monitor and the Estimated Available Bandwidth (EABW) Monitor -- are the most complicated. They operate as separate threads within the application, periodically monitoring key statistics (in the case of the CPU monitor, key statistics generated by a separate application) and updating a structure in the DCFD containing the current value of the Monitor. When a Model needs information from one of these Monitors, it locks the value in the structure for reading, reads the current value, and releases the lock.

In the process of developing Datacomp, I created these Monitors outside of any particular Model and wanted to be able to execute them while using any Model in order to obtain operational and debugging data. This required starting the Monitors by Datacomp during `make_dcfd()`. Similarly, outputs for these Monitors are written into special portions of the DCFD reserved for their use, rather than into Model-specific locations. At the same time, during some tests I wanted to disable any extraneous code, so Datacomp has the option to skip loading these Monitors at start time.

In a more mature version of Datacomp meant for broader use, it makes organizational sense for Monitors of this type to be started and stopped by the Model's initialization and destructor functions. This would make the code and operation cleaner; Monitors would only be started by a Model if required. Also, rather than storing Monitor data in special portions of the DCFD, it would be stored in the Model's scratch space. However, this requires the development of a formal API for Monitors (specifying things like output location and so on), so that Monitor code could be included in any Model. EABW and CPU monitors in the current version of

Datacomp are nevertheless modular enough that they can be changed independently of other Datacomp components.

APPENDIX B: SAMPLE DCSF DATA

The following hex dump (prepared with `xxd`) is a binary example of `dzip`-compressed data.

```
0000000: bbc0 dada 1000 0000 0000 0000 1000 0000  ................
0000010: efbe edfe 0000 0000 0000 0000 0000 0000  ................
0000020: bbc0 dada a504 0000 0200 0200 a602 0000  ................
0000030: 78da 8554 4b6f d340 103e e35f 31e4 d44a  x..TKo.@.>._1..J
0000040: 8d4b a904 a287 48c0 85d0 1e2a 5ab8 4fec  .K....H....*Z.O.
0000050: 71bc 64bd e3ee a396 ff3d dfd8 8513 8228  q.d......=.....(
0000060: 729c ec3c be97 c33e d79d fa93 d728 75a3  r..<...>.....(u.
0000070: c358 b2c4 44db 2d5d bdb9 bcbe bafc f0ae  .X..D.-]........
0000080: faca 8324 dadc ba83 6ee8 9e63 9ce9 5555  ...$....n..c..UU
0000090: ede9 6749 993a 2da1 252d 9972 cf99 920e  ..gI.:-.%-.r....
00000a0: a241 4843 2334 7028 1d37 b944 6989 e924  .AHC#4p(.7.Di..$
00000b0: f341 39b6 34f5 aee9 69d2 e2db 4a9b 8693  .A9.4...i...J...
00000c0: d3c0 decf 9423 8734 389b 2556 4e49 9e8a  .....#.48.%VNI..
00000d0: d8a8 cd24 1ee8 845a ce6c 3037 9475 29fb  ...$...Z.l07.u).
00000e0: 8db9 e20e 57ac 1925 3a05 a48e 5cc0 72f7  ....W..%:...\.r.
00000f0: ecf2 5cd3 d99e 26e7 3d4d 7c44 55d0 83b6  ..\...&.=M|DU...
0000100: 189f eda7 24de 279b 95a4 3ea7 07a5 7d35  ....$.'...>...}5
0000110: 49c8 b6e0 a3cf fcec 5266 fb92 4400 5d31  I.......Rf..D.]1
0000120: 02d5 f3e8 c211 1b30 a374 1d66 e79e 12bb  .......0.t.f....
0000130: f60f cb9b aada 3df6 2e11 de4c d118 3394  ......=....L..3.
0000140: 8a32 e1d0 7a0f aaa7 8be5 37a6 0152 6e7b  .2..z.....7..Rn{
0000150: 7e16 e819 0d4a b5b3 e3d4 8bef 8c09 1bf2  ~....J..........
0000160: de20 805b 49de a017 10b8 87b4 7467 333e  . .[I.......tg3>
0000170: 95b6 c5b6 a1a6 47eb 1ea3 1ebc 0c69 7565  ......G......iue
0000180: 4f3d b72b 4688 3b49 346f 5c58 c6cc 1efc  O=.+F.;I4o\X....
0000190: 5c73 635b 233a 1987 59d5 cc33 fd43 36fd  \sc[#:..Y..3.C6.
00001a0: 5fe6 8877 ad80 34c4 28a1 81ab 2ec3 d9a7  _..w..4.(.......
00001b0: a299 333c 4c17 6807 16b3 a583 f81e e68c  ..3<L.h.........
00001c0: 332e 309a 1676 0791 804b 36a7 0051 bb28  3.0..v...K6..Q.(
00001d0: 0076 b6ac 26b6 f54b a824 6839 f634 b894  .v..&..K.$h9.4..
00001e0: 4698 03b5 9219 7080 fd00 1bcd d570 5cb5  F.....p......p\.
00001f0: b35d c39c f22a 0c88 f791 d35f f3c2 e328  .]...*....._...(
0000200: 8c68 6ba0 1121 a0b7 efaf cf6b e439 511b  .hk..!.....k.9Q.
0000210: 794a 0be1 6a67 b51c 5d02 9b55 b0cf 7d34  yJ..jg..]..U..}4
0000220: 8118 5dcb 6284 71d1 c274 0a8a 4f3f f19c  ..].b.q..t..O?..
0000230: 5ed2 88fd 0a79 1664 ad7b 9688 5b90 0da6  ^....y.d.{..[...
0000240: 0e31 0aa9 8b3a aca0 6100 861c cb80 e39a  .1...:..a.......
0000250: bee8 24a8 8782 a920 2a2e 2074 8bf0 b6e7  ..$.... *. t....
0000260: a940 688a b85d 49ef 2d0c 5e7a f081 01b0  .@h..]I.-.^z....
0000270: 310a 7063 505b edb2 65ce c2b3 44e9 18b9  1.pcP[..e...D...
0000280: 2d8c 66cb 0b16 a5a5 3f35 6887 1675 553d  -.f.....?5h..uU=
0000290: ae4f dafa 6002 119e b7b4 a0b5 944f d119  .O..`........O..
00002a0: 7a2a a3a9 f663 ffed fbc3 f66e 9960 32d5  z*...c.....n.`2.
00002b0: 10e6 94aa 3c29 cd8b b27c 544c a47f bef0  ....<)...|TL....
00002c0: c772 5bff a706 f438 04a6 4399 b157 5e57  .r[....8..C..W^W
00002d0: bf00 9074 993b                            ...t.;
```

APPENDIX C: USING AND CONTRIBUTING TO THIS PROJECT

I believe in Adaptive Compression as a means of improving efficiency and I hope to continue to do research in the area. I also believe in (and have benefitted greatly from) Free Software, and I think that research paid for with public dollars should be made public whenever it is practical to do so. Finally, I like the idea that a project I spent so much time on might be useful for others and not sit on a forgotten hard disk somewhere.

Because of these things, I tried to design Datacomp and Comptool to be useful for others as platforms for research into Adaptive Compression. Accordingly, I plan to release the source code and selected data from my tests under an open source license along with documentation for the tools. In order to facilitate others using Datacomp and Comptool, I hope to support a source repository, mailing list and so on.

The future home of this work will be:

http://labs.tastytronic.net/datacomp/

It may take me some time to organize materials for a first public release. If you are interested in this work, I invite you to check out the website or contact me for more information at pedro@tastytronic.net.

14 Bibliography

[1] John of Salisbury, The Metalogicon of John of Salisbury. Trans. Daniel D. McGarry, University of California Press, 1955.

[2] Author Unknown, "ssh(1): OpenSSH client Linux man page," 14 04 2013. [Online]. Available: http://linux.die.net/man/1/ssh. [Accessed 06 07 2013].

[3] K. Barr and K. Asanovic, "Energy-aware lossless data compression," *ACM Trans. Comput. Syst.,* vol. 24, no. 3, pp. 250-291, 2006.

[4] Google, Inc., "Data Compression Proxy - Google Chrome Mobile," 03 2013. [Online]. Available: https://developers.google.com/chrome/mobile/docs/data-compression. [Accessed 06 07 2013].

[5] G. Pandya, "Full Disclosure: Nokia phone forcing traffic through proxy (mailing list)," 12 2012. [Online]. Available: http://seclists.org/fulldisclosure/2012/Dec/95. [Accessed 06 07 2013].

[6] Opera Software, "Opera Help: Opera Turbo," [Online]. Available: http://help.opera.com/Linux/10.60/en/turbo.html. [Accessed 06 07 2013].

[7] D. Salomon, Data Compression: The Complete Reference 4th Edition, Springer, 2007.

[8] P. Deutsch, "RFC 1952 - GZIP file format specification version 4.3," 05 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1952.txt. [Accessed 06 07 2013].

[9] J. Seward, "bzip2: Home," 2013. [Online]. Available: http://www.bzip.org/. [Accessed 06 07 2013].

[10] MaximumCompression.com, "Compression Programs / Software," 2011. [Online]. Available:

http://www.maximumcompression.com/programs.php. [Accessed 26 June 2013].

[11] R. M. Fano, "The transmission of information," Technical Report No. 65, Research Laboratory of Electronics at MIT, Cambridge, MA, USA, 1949.

[12] D. Huffman, "A method for the construction of minimum-redundancy codes," in *Proceedings of the IRE*, 1952.

[13] M. Wheeler and D. Burrows, *A block-sorting lossless data compression algorithm,* Palo Alto, CA: DEC Systems Research Center, 1994.

[14] Author Unknown, "Wikipedia article "bzip2 : Compression Stack" (confirmed via correspondence with bzip2 author Julian Seward)," 17 07 2013. [Online]. Available: http://en.wikipedia.org/wiki/Bzip2#Compression_stack. [Accessed 17 07 2013].

[15] J. Lempel and A. Ziv, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory,* vol. 23, no. 3, pp. 337-343, 1977.

[16] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," *ACM Transactions on Information Theory,* vol. 24, no. 5, pp. 530-536, 1978.

[17] T. Welch, "A Technique for High-Performance Data Compression," *IEEE Computer,* pp. 8-19, 06 1984.

[18] CompuServe Incorporated, "Graphics Interchange Format Version 89a," 31 07 1990. [Online]. Available: http://www.w3.org/Graphics/GIF/spec-gif89a.txt. [Accessed 07 07 2013].

[19] D. Marti, "Burn All GIFs," 26 09 2013. [Online]. Available: http://burnallgifs.org/. [Accessed 26 09 2013].

[20] P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," 05 1996. [Online]. Available: http://tools.ietf.org/rfc/rfc1951.txt. [Accessed 06 07 2013].

[21] M. Oberhumer, "oberhumer.com: LZO real-time data compression library," 12 08 2011. [Online]. Available: http://www.oberhumer.com/opensource/lzo/. [Accessed 07 07 2013].

[22] I. Pavlov, "LZMA SDK (Software Development Kit)," 2013. [Online]. Available: http://7-zip.org/sdk.html. [Accessed 07 07 2013].

[23] G. Nigel and N. Martin, "Range encoding: an algorithm for removing redundancy from a digitised message," in *Video & Data Recording Conference*, Southampton, 1979.

[24] Author Unknown, "bzip2 -- Wikipedia, the free encyclopedia," 6 07 2013. [Online]. Available: http://en.wikipedia.org/wiki/Bzip2. [Accessed 23 08 2013].

[25] J. Seward, "bzip2 man page," [Online]. Available: http://www.bzip.org/1.0.5/bzip2.txt. [Accessed 07 07 2013].

[26] Apache Software Foundation, "mod_deflate - Apache HTTP Server," 2013. [Online]. Available: http://httpd.apache.org/docs/2.0/mod/mod_deflate.html. [Accessed 28 06 2013].

[27] OpenSSH, "OpenSSH Homepage," 19 June 2013. [Online]. Available: http://www.openssh.com/. [Accessed 19 June 2013].

[28] A. Shacham, B. Monsour, R. Pereira and M. Thomas, "RFC 3173: IP Payload Compression Protocol (IPComp)," 2001 September. [Online]. Available: http://tools.ietf.org/rfc/rfc3173.txt. [Accessed 06 07 2013].

[29] J. Yonan, "OpenVPN 2.0.x Manpage," 2010. [Online]. Available: http://openvpn.net/index.php/open-source/documentation/manuals/65-openvpn-20x-manpage.html. [Accessed 27 June 2013].

[30] M. Perdeck, "Making the most out of IIS compression...," 24 Aug 2011. [Online]. Available: http://www.codeproject.com/Articles/242133/Making-the-most-out-of-IIS-compression-Part-1-

conf#caching-compressed-dynamic. [Accessed 28 06 2013].

[31] P. Politopoulos, E. Markatos and S. Ioannidis, "Evaluation of compression of remote network monitoring data streams," in *IEEE Network Operations and Management Symposium Workshops*, 2008.

[32] A. I. Pavlov, "SquashFS HOWTO," 24 July 2008. [Online]. Available: http://www.tldp.org/HOWTO/html_single/SquashFS-HOWTO/. [Accessed 19 June 2013].

[33] Microsoft Incorporated, "What is DoubleSpace and How Does It Work?," [Online]. Available: http://technet.microsoft.com/en-us/library/cc722457.aspx. [Accessed 17 07 2013].

[34] F. Douglis, "On the role of compression in distributed systems," *SIGOPS Oper. Syst. Rev.,* vol. 27, no. 2, pp. 88-93, 1993.

[35] A. de Maricourt, "e2compr - transparent compression for the ext2 file system," [Online]. Available: http://e2compr.sourecorge.net/. [Accessed 19 June 2013].

[36] O. Rodeh and A. Teperman, "zFS - a scalable distributed file system using object disks," *Mass Storage Systems and Technologies,* pp. 207-218, 2003.

[37] O. Rodeh, J. Bacik and C. Mason, "BTRFS: The Linux B-tree Filesystem," IBM Research, San Jose, CA, 2012.

[38] Author Unknown, "Btrfs Wiki," [Online]. Available: https://btrfs.wiki.kernel.org/index.php/Main_Page. [Accessed 19 June 2013].

[39] L. Yang, H. Lekatsas and R. P. Dick, "High-performance operating system controlled memory compression," in *Proceedings of the 43rd Annual Design Automation Conference*, 2006.

[40] N. Gupta, "Compcache: in-memory compressed swapping," May 2009. [Online]. Available: http://lwn.net/Articles/334649/. [Accessed 19 June 2013].

[41] F. Douglis, "The compression cache: Using on-line compression to extend physical memory.," in *Proceedings of 1993 Winter USENIX Conference*, 1993.

[42] A. Appel and K. Li, "Virtual memory primitives for user programs," in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, Santa Clara, CA, 1991.

[43] L. Yang, R. P. Dick, H. Lekatsas and S. Chakradhar, "Online memory compression for embedded systems," *ACM Trans. Embed. Comput. Syst.,* vol. 9, no. 3, 2010.

[44] R. S. de Castro, A. Lago and D. Da Silva, "Adaptive Compressed Caching: Design and Implementation," in *Computer Architecture and High Performance Computing, Symposium on*, Los Alamitos, CA, 2003.

[45] T. Berners-Lee, R. Fielding and H. Frystyk, May 1996. [Online]. Available: http://tools.ietf.org/html/rfc1945. [Accessed 23 September 2013].

[46] R. Kothiyal, V. Tarasov, P. Sehgal and E. Zadok, "Energy and Performance Evaluation of Lossless File Data Compression on Server Systems," in *Israeli Experimental Systems Conference (ACM SYSTOR '09)*, Haifa, Israel, 2009.

[47] R. Kothiyal, "Energy and Performance Evaluation of Lossless File Data Compression on Computer Systems (M.S. Thesis)," Stony Brook University (TR FSL-09-02), 2009.

[48] W. Culhane, "Statistical Measures as Predictors of Compression Savings," The Ohio State University, 2008.

[49] C. Pu and L. Singaravelu, "Fine-grain adaptive compression in dynamically variable networks," in *Proceedings IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2005.

[50] M. Gray, P. Peterson and P. and Reiher, "Scaling Down Off-The-Shelf Data Compression:

Backwards-Compatible Fine-Grain Mixing," in *IEEE International Conference on Distributed Computing Systems*, Macau, 2012.

[51] R. Fielding, G. J., J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, "RFC 2616 Hypertext Transfer Protocol -- HTTP/1.1," June 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2616.txt. [Accessed 06 07 2013].

[52] L. Wang and J. Manner, "Evaluation of data compression for energy-energyaware communication in mobile networks," in *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, 2009.

[53] S. Klein and Y. Wiseman, "Parallel Lempel Ziv coding," *Discrete Applied Mathematics,* vol. 146, no. 2, pp. 180-191, 2005.

[54] Intel Corporation, "SA-110 Microprocessor," Intel Corporation, 1998.

[55] B. Knutsson and M. Bjorkman, "Adaptive end-to-end compression for variable-bandwidth communication," *Computer Networks,* vol. 31, no. 7, pp. 767-779, 1999.

[56] C. Krintz and B. Calder, "Reducing delay with dynamic selectino of compression formats," in *Proceedings of IEEE International Symposium on High Performance Computing*, 2001.

[57] Oracle Corporation, "JAR File Specification," 2011. [Online]. Available: http://docs.oracle.com/javase/6/docs/technotes/guides/jar/jar.html. [Accessed 07 07 2013].

[58] W. Pugh, "Compressing Java class files," in *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*, 1999.

[59] N. Motgi and A. Mukherjee, "Network conscious text compression system (NCTCSys)," in *Proceedings of the IEEE International Conference on Information Technology: Coding and Computing*, 2001.

[60] F. S. Awan and A. Mukherjee, "LIPT: A Lossless Text Transform to Improve Compression," in *Proceedings of the IEEE International Conference on Information Technology: Coding and Computing*, 2001.

[61] S. Sucu and C. Krintz, "Ace: A resource-aware adaptive compression environment," in *Proceedings of IEEE International Conference on Information Technology: Coding and Computing*, 2003.

[62] C. Krintz and S. Sucu, "Adaptive on-the-fly compression," *IEEE Transactions on Parallel and Distributed Systems,* vol. 17, no. 1, pp. 15-24, 2006.

[63] R. Xu, Z. Li, C. Wang and P. Ni, "Impact of data compression on energy consumption of wireless-networked handheld devices," in *23rd International Conference on Distributed Computing Systems*, 2003.

[64] Y. Wiseman, K. Schwan and P. Widener, "Efficient end to end data exchange using configurable compression," *ACM SIGOPS Operating Systems Review,* vol. 39, no. 3, pp. 4-23, 2005.

[65] R. P. Brent, "A linear algorithm for data compression.," *Australian Computer Journal,* vol. 19, no. 2, pp. 64-68, 1987.

[66] E. Jeannot, "Improving Middleware Performance with AdOC: an Adaptive Online Compression Library for Data Compression," in *19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[67] M. Lehmann, "Marc Lehmann's "LibLZF"," 25 08 2008. [Online]. Available: http://oldhome.schmorp.de/marc/liblzf.html. [Accessed 27 06 2013].

[68] R. Maddah and S. Sharafeddine, "Energy-aware adaptive compression scheme for mobile-to-mobile communications.," in *IEEE International Symposium on Spread Spectrum Techniques and*

Applications, 2008.

[69] Author Unknown, ".NET Zip Library #ziplib (SharpZibLib)," [Online]. Available: http://www.icsharpcode.net/opensource/sharpziplib/. [Accessed 07 07 2013].

[70] P. Trimintzios, M. Polychronakis, A. Papadogiannakis, M. Foukarakis, E. Markatos and A. Oslebo, "DiMAPI: An application programming interface for distributed network monitoring," in *10th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2006.

[71] L. S. Tan, S. P. Lau and C. E. Tan, "Quality of Service Enhancement via Compression Technique for Congested Low Bandwidth Network," in *IEEE Malaysia International Conference on Communications (MICC)*, 2011.

[72] S. Chen, S. Ranjan and A. Nucci, "IPzip: A stream-aware IP compression algorithm," in *IEEE Data Compression Conference (DCC)*, 2008.

[73] M. Shimamura, T. Ikenaga and M. Tsuru, "Compressing Packets Adaptively Inside Networks," *IEICE Transactions on Communications,* vol. 93, no. 3, pp. 501-515, 2010.

[74] Author Unknown, "The Network Simulator - ns2," [Online]. Available: http://www.isi.edu/nsnam/ns/. [Accessed 18 07 2013].

[75] Y. Xiao, M. Siekkinen and A. Yla-Jaaski, "Framework for Energy-aware Lossless Compression in Mobile Services: the Case for E-mail," in *IEEE International Conference on Communications*, 2010.

[76] E. Roshal, "WinRAR archiver, a powerful tool to process RAR and ZIP files," win.rar GmbH, [Online]. Available: http://www.rarlab.com/. [Accessed 07 07 2013].

[77] B. Nicolae, "On the benefits of transparent compression for cost-effective cloud data storage," *Transactions on large-scale data-and-knowledge-centered systems III,* pp. 167-184, 2011.

[78] K. W. Park and K. H. Park, "ACCENT: Cognitive cryptography plugged compression for SSL/TLS-based cloud computing," *ACM Transactions on Internet Technology,* vol. 11, no. 2, pp. 2-31, 2011.

[79] M. Geelnard, "Basic Compression Library -- an open source compression library (Huffman, RLE, LZ77, Rice, Shannon-Fano)," 22 07 2006. [Online]. Available: http://bcl.comli.eu. [Accessed 07 07 2013].

[80] T. Dierks and E. Rescoria, "The Transport Layer Security (TLS) Protocol Version 1.2," 08 2008. [Online]. Available: http://tools.ietf.org/rfc/rfc5246.txt. [Accessed 07 07 2013].

[81] M. Hovestadt, O. Kao, A. Kliem and D. Warneke, "Evaluating Adaptive Compression to Mitigate the Effects of Shared I/O in Clouds," in *IEEE Internation Parallel & Distributed Processing Symposium*, 2011.

[82] Author Unknown, "Fast compression library for C, C# and Java," [Online]. Available: http://www.quicklz.com/. [Accessed 07 07 2013].

[83] D. Warneke and O. Kao, "Nephele: Efficient parallel data processing in the cloud," in *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, 2009.

[84] D. Singh and W. Kaiser, "Energy Efficient Network Data Transport through Adaptive Compression using the DEEP Platforms," in *Wireless and Mobile Computing, Networking and Communications*, 2012.

[85] D. Singh, P. Peterson, P. Reiher and W. J. Kaiser, "The Atom LEAP Platform For Energy-Efficient Embedded Computing: Architecture, Operation, and System Implementation," UCLA, 2010.

[86] A. Tridgell, "rzip," [Online]. Available: http://rzip.samba.org/. [Accessed 07 07 2013].

[87] T. Buennemeyer, T. Nelson, R. Marchany and J. Trong, "Polling the smart battery for efficiency:

lifetime optimization in battery-sensing intrusion protection systems," in *Proceedings of SoutheastCon*, 2007.

[88] M. Cierniak, G. Lueh and J. Stichnoth, "Practicing JUDO: Java under Dynamic Optimization," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[89] R. Wolski, "Dynamically forecasting network performance using the network weather service," *Cluster Computing,* vol. 1, no. 1, pp. 119-132, 1998.

[90] SpecWeb2009. [Online].

[91] TCPDUMP Contributors, "TCPDUMP/LIBPCAP public repository," [Online]. Available: http://www.tcpdump.org/. [Accessed 06 07 2013].

[92] G. Combs, "Wireshark -- Go Deep," [Online]. Available: http://www.wireshark.org/. [Accessed 23 07 2013].

[93] B. Gregg, "Chaosreader," [Online]. Available: http://chaosreader.sourceforge.net/. [Accessed 06 07 2013].

[94] P. Peterson, *Datacomp: Transparently Improving Efficiency with Adaptive Compression (Ph.D. Prospectus),* 2011.

[95] Electronic Educational Devices, *WattsUp? PRO Operators Manual,* Denver, CO, 2003.

[96] M. Cracauer, "cstream - a general purpose streaming tool," 2012. [Online]. Available: http://www.cons.org/cracauer/cstream.html. [Accessed 06 07 2013].

[97] J. Kleinberg and E. Tardos, Algorithm Design, Pearson Education, Inc., 2006.

[98] Various, "GlobalInterpreterLock - Python Wiki," 04 09 2013. [Online]. Available: https://wiki.python.org/moin/GlobalInterpreterLock. [Accessed 27 09 2013].

[99] Alexa, Inc., "Alexa Top 500 Global Sites," Alexa, Inc, 18 07 2013. [Online]. Available:

http://www.alexa.com/topsites. [Accessed 18 07 2013].

[100] G. Combs, "Wireshark - Go Deep.," 06 07 2013. [Online]. Available: http://www.wireshark.org/. [Accessed 06 07 2013].

[101] E. Rescoria, "RFC 2818 -- HTTP Over TLS," May 2000. [Online]. Available: http://tools.ietf.org/rfc/rfc2818.txt. [Accessed 06 07 2013].

[102] S. Yegulalp, "Hacking Firefox: The secrets of about:config," 29 May 2007. [Online]. Available: http://www.computerworld.com/s/article/9020880/Hacking_Firefox_The_secrets_of_about_config. [Accessed 06 07 2013].

[103] TIS Committee, "Executable and Linking Format (ELF)," May 1995. [Online]. Available: http://refspecs.linuxbase.org/LSB_3.1.1/LSB-Core-generic/LSB-Core-generic/elf-generic.html. [Accessed 06 07 2013].

[104] E. Hall, "RFC 4155 - The application/mbox Media Type," 09 2005. [Online]. Available: https://tools.ietf.org/rfc/rfc4155.txt. [Accessed 06 07 2013].

[105] D. J. Bernstein, "Using maildir format," [Online]. Available: http://cr.yp.to/proto/maildir.html. [Accessed 06 07 2013].

[106] P. Peterson, "surf the web -- FOR SCIENCE!," Tastytronic Industries, 24 03 2013. [Online]. Available: http://tastytronic.net/~pedro/for_science/. [Accessed 06 07 2013].

[107] B. Hubert, T. Graf, G. Maxwell, R. van Mook, M. van Oosterhout, P. Schroeder, J. Spaans and P. Larroy, "Linux Advanced Routing & Traffic Control HOWTO," 20 05 2012. [Online]. Available: http://www.lartc.org/. [Accessed 06 07 2013].

[108] B. Hubert, "Wonder Shaper," 2002. [Online]. Available: http://lartc.org/wondershaper/. [Accessed 06 07 2013].

[109] M. Carbone and L. Rizzo, "Dummynet Revisited," *ACM SIGCOMM Computer Communication Review,* vol. 40, no. 2, pp. 12-20, 2010.

[110] E. Kohler, R. Morris, B. Chen, J. Jannotti and M. F. Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems,* vol. 18, no. 3, pp. 263-297, 2000.

[111] A. Wood, "ivarch.com: Pipe Viewer," 22 01 2013. [Online]. Available: http://www.ivarch.com/programs/pv.shtml. [Accessed 06 07 2013].

[112] Author Unknown, *XZ Manpage for Ubuntu Linux,* Ubuntu Linux, 2013.

[113] National Instruments, "Top Eight Features of the Intel Core i7 Processors for Test, Measurement and Control," 2011. [Online]. Available: http://www.ni.com/white-paper/11266/en/pdf. [Accessed 06 07 2013].

[114] S. Dawson-Haggerty, A. Krioukov and D. Culler, "Power Optimizatoin -- A Reality Check," University of California, Berkeley, Berkeley, CA, 2009.

[115] S. Josefsson, "RFC 4648: The Base16, Base32, and Base64 Data Encodings," 10 2006. [Online]. Available: https://tools.ietf.org/html/rfc4648. [Accessed 17 08 2013].

[116] C. Fleming, P. Peterson, E. Kline and P. Reiher, "Data Tethers: Preventing Information Leakage by Enforcing Environmental Data Access Policies," in *International Conference on Communications (ICC)*, 2012.

[117] P. Peterson, "Cryptkeeper: Improving Security with Encrypted RAM," in *IEEE Conference on Homeland Security Technologies*, 2010.