Datacomp: Locally Independent Adaptive Compression for Real-World Systems

Peter A. H. Peterson*, Peter L. Reiher†
*University of Minnesota Duluth
pahp@d.umn.edu
†University of California, Los Angeles
reiher@cs.ucla.edu

***Abstract*—Non-lossy compression can save time and energy during communication if the cost to compress and send input is less than the cost of sending it uncompressed. Unfortunately, compression can also degrade performance; no single method is always beneficial, and outcomes depend on many factors. As a result, compression choices in real systems are coarsely grained and manually controlled, resulting in suboptimal or even poor performance. Adaptive Compression (AC) systems make compression choices dynamically to optimize utility. Existing AC systems are limited in ways that reduce their suitability for general-purpose computers. Datacomp is an AC system that operates locally and includes no significant hard-coded knowledge. Using real-world data, a broad range of environments and the Comptool "AC Oracle," we show that Datacomp's performance is equivalent or close to the ideal at bandwidths between 1-100Mbit/s, even when static strategies are suboptimal or more costly than no compression. While Datacomp struggles to perform well at 1Gbit/s, understanding why illustrates important challenges for AC systems and suggests solutions.**

I. Introduction

Compression can improve the efficiency of communication when data is compressible and bandwidth, not computation, is the performance bottleneck. Communication cost is a function of size, which compression can reduce. Additionally, computers can generally perform many operations during the time it takes to transmit a single bit[2] over a network, and decompression is typically cheaper than compression. Thus, if the input can be compressed quickly and significantly enough, it can increase the *effective throughput* of the channel, using less time and energy than sending the data uncompressed. However, some data is not compressible and can even expand when compressed; effort spent compressing it is wasteful.

Furthermore, being compressible does not guarantee that compression will be profitable, because improving communication efficiency with compression requires meeting or exceeding size and run time requirements that are situation-dependent and may not be possible. In other words, the compressor must be fast and effective enough *in* the current execution environment, *on* the given data and *with respect to* the I/O channel in order to save more than it costs. Unfortunately, it is generally not known *a priori* if compression will be profitable on some input with any method in the current execution and communication environment. A given compressor might improve throughput in one scenario but degrade it in another if the data is less compressible, the available bandwidth (ABW) has increased, or there is less available CPU time.

Due to the risk of degrading performance, most systems and applications do not compress or they use a specific method only when it is near certain to improve (or at least not degrade) performance. If compression is instead under manual control, it is typically limited to a per-session choice of *on* or *off*, which is suboptimal if conditions are sufficiently dynamic. Furthermore, even if compression could be enabled and disabled at will, it would often be impossible for a human to achieve optimal results due to the frequency of decisions required. As a result, the *status quo* captures low-hanging fruit and generally avoids actively wasting resources, but ignores many situations that could profit from compression.

Computers excel at high-speed analysis and decision-making. Adaptive Compression (AC) centers on developing mechanisms to dynamically choose and apply profitable compression choices on the fly to save resources such as time, space or energy. AC systems attempt to choose the best compression method for every situation, with the assumption that the best "method" may be to not compress at all. By choosing the best method at every opportunity, AC systems reap the benefits of compression when they exist, and avoid losses when they do not.

This paper describes the design and evaluation of Datacomp – a locally independent AC system for individual, general-purpose computing devices in realistic environments. While previous AC systems have achieved significant efficiency improvements, they have limitations that make their performance in general computing environments unclear. In contrast, Datacomp uses no remote support, makes no workload assumptions, includes almost no precomputed or hard-coded information and makes decisions using a local learning model.

We demonstrate significant improvements in many realistic scenarios by using Datacomp to perform network transmission of nine classes of data in a range of environments. Additionally, we show that Datacomp's strategies are close to the ideal. These results suggest that the efficiency of general-purpose computing devices could be significantly improved through the use of locally independent AC mechanisms like Datacomp.

II. Adaptive Compression

While compression can often improve throughput, no single compression method is ideal for all scenarios. As a result, unless the workload is sufficiently static, choosing to universally apply some static method is at best suboptimal and at worst can degrade performance. Adaptive Compression (AC) systems attempt to optimize performance by dynamically choosing and using the best available method at every opportunity. This is a challenging goal because the best choice depends on many dynamic, interrelated and hard-to-predict factors.

All AC systems attempt to optimize some value by identifying and applying the best available method for the current opportunity. Because of this fundamental similarity, AC systems share a number of abstract components that we call *methods*, *monitors*, *models* and *mechanisms*.

A. Compression Methods

A *method* is a distinct compression mechanism. Many different compressors are used by AC systems, including LZO[17], zlib[1], bzip2[23] and xz[21]. This diversity is due to differing compression trade-offs and provides meaningful alternatives for AC decisions. LZO is much faster than gzip but typically compresses less effectively. Xz can compresses more effectively than LZO or gzip, but can be slow. One compressor may provide multiple distinct methods; for example, zlib supports a "strength level" that changes the balance between compression and run time. Typically, each AC system chooses between at least two methods: "null compression" (copying the data) and one or more actual compressors. Some systems use multiple compression libraries[24], [20], modulate strength levels of a single algorithm[13] or both[11].

A common way to describe compression effectiveness is in terms of Compression Ratio (CR), the ratio of compressed bytes divided by the input size. The reciprocal is the Compression Gain (CG) – the amount by which compressed data will expand when decompressed. When compressing to improve throughput, another important metric is the Output Rate (OR) – the rate at which a method emits compressed bytes. 100MB compressed to 80MB in 1.0 seconds results in a OR of 80MB/s, a CR of 0.80 and a CG of 1.25. In general, the CR and CG of a particular operation depends on many factors. The exception is "null compression" (NC), which always achieves a CR and CG of 1.0 and the maximum output rate, since it merely copies data.

The available bandwidth (ABW) of an I/O channel is the rate at which bytes can be transferred. The Effective Output Rate (EOR) is the rate at which *information* is being transmitted over a compressed channel regardless of the number of bytes used. As compression can enable the same information to be sent using fewer bytes in less time, it can result in an EOR that is greater than the ABW.

If the ABW is greater than the OR, the EOR for a method is its OR times its compression gain $(OR * CG)$. However, if the method's OR is *greater* than the ABW, then the method's rate is limited to *at most* the ABW. In this case, the EOR is instead $(ABW * CG)$ – the amount of data that can be sent over the I/O channel multiplied by the gain specific to the method used. Combining these two cases, we have:

$$EOR = min(ABW, OR) * CG \tag{1}$$

Given that each method has different performance properties, the goal of AC is to choose the method that will result in the greatest EOR. However, this is complicated by two main factors: ABW and input data are often dynamic and the CR and CG of methods are data dependent.

B. Compression Opportunities

A compression *opportunity* is the unique set of performance-determining conditions that exist when an AC decision must be made, and for which one or more methods are ideal. Opportunities are determined by the properties of the *data* and *environment*. Properties of the data affect OR, CR, and CG as compressibility varies by content and method. More input generally improves CR and OR.

Properties of the environment are also critical because of the time-sensitive nature of AC. The ABW is the raw rate that AC tries to effectively increase via compression. As bandwidth increases, profiting from AC becomes harder because transmission costs decrease relative to computation. Execution environment properties that could affect a method's output rate (OR) are also essential, including CPU load, frequency, number of cores, available memory, etc. Combined, the data and environment determine the opportunity problem space for which the best method is the solution. Assuming that all critical properties of an opportunity are identified, performance for a compressor in a given opportunity should be constant because compressor behavior is deterministic.

C. Monitors, Models and Mechanisms

Monitors are the modules responsible for obtaining decision information, including data and environment properties and the results of choices. Monitors can be simple (e.g., reading a value), or complex (e.g., predicting values). AC *models* make choices using monitored values as input. Finally, these components are united by a *mechanism* that determines the abstraction layer at which the system operates. Examples include application-level (libraries [11], [20] or built-in [15]), remote proxies [8], [19], [18], a Java virtual machine [24], middleware[14] and the kernel[13]. The mechanism also defines the API of the system and the protocol used to encapsulate adaptively compressed data.

D. Related Work

Barr and Asanovic's influential study[2] showed that (on their platform) roughly 1,000 computations could be performed on one bit for the cost of sending it across a network. In practice, computational throughput usually far outstrips I/O bandwidth. This surplus results in a window of opportunity within which compression can be used to improve efficiency.

Datacomp is similar in spirit to several previous AC systems. Knutsson and Bjorkman implemented an early system that adaptively compressed data in kernel send buffers by

varying the zlib strength level based on the rate that the buffers were being drained.[13] Jeannot expanded that work in AdOC, a pipe-based user-level library.[11] Krintz and Sucu built the Java virtual machine-based ACE[24], which chooses between multiple methods using a precomputed method comparison model, recent history and the Network Weather Service[27]. Remote compression proxies[29], [8], [19], [18] can improve the efficiency of end-hosts by performing compression in the network. AC systems have also been built into in high-performance distributed computing middleware[28], [14], [16]. Finally, ACCENT[20] added AC capabilities to an SSL library using a model that made AC choices while considering the additional cost of encryption.

Some systems have dependencies that, while appropriate for their intended use, limit their suitability for the general computing environment; grid middleware, trusted network monitors and external computation support can all facilitate AC, but the average laptop or mobile device cannot depend on these facilities being available. Furthermore, external dependencies transfer AC costs to other systems rather than directly improving end host efficiency, which may actually increase global resource use. Additionally, compression proxies have other drawbacks: they only improve download throughput (since the proxy is remote), and they raise security and privacy concerns due to their access to raw input.[19]

Many systems also rely on hard-coded assumptions or precomputed models. While efficient, these models reflect the platform, input, compression methods, system state and intuition used to create them and may not be accurate in other circumstances. One approach to estimate CR uses a table of averages by file extension, but this may not be accurate for file types with highly variable CRs (e.g., PDFs and binaries) and requires extensions to be available. Another common assumption is that it is worthwhile to spend surplus CPU time on increased compressor "strength," but in practice higher strength levels do not always improve CR (and may reduce OR). The authors of ACE precomputed a method comparison model based on an observed linear relationship between the average CR and run-time of their methods on their platform. However, this approach is not only hardware and data dependent, but it does not recognize that some compressors are non-linearly more effective for specific types of data (see Section V-C). In contrast, Datacomp recognizes differences in data type irrespective of method or source, makes no assumptions about relationships between methods, builds an adaptive model based on the local hardware and encountered data and generally avoids hard-coded choices based on intuition.

In addition, some systems were designed or tested in such a way that it is difficult to estimate their behavior in more common environments. For example, many previous workloads are not very diverse nor are they similar to typical user data. Several systems were specifically designed for an intended workload, such as improving throughput for large file transfers[11] or specific types of data (e.g., VM images[16]). Some systems have conservative minimum sizes for compression (e.g., 32KB[24], 80KB or 512KB[11] and 2MB or 64MB[16]), which simplifies prediction and improves CR and OR, but would result in many real-world payloads being ignored even though they might benefit from AC. In contrast, Datacomp considers compressing even very small inputs, does not make assumptions about data characteristics, and was evaluated using diverse workloads composed of a large corpus of realistic data in many environmental conditions.

Finally, most AC systems monitor the characteristics of input in some way. ACE[24] uses what we call the "last block assumption" (LBA), which assumes that current and previous inputs are similar. While effective, the LBA offers no guidance for the first choice (which may be the only choice for small transfers), can leave the system "one step behind" if data properties fluctuate, and struggles with long segments of incompressible data. Other systems compress a sample of the input to estimate CR and OR,[28], [16] but this is wasteful if the compressed output is discarded and inaccurate if the performance of the sampling method is dissimilar from other methods. In contrast, Datacomp uses a fast non-compressing function that estimates input compressibility. Along with sampling techniques, our novel mechanism (similar in spirit to work by Culhane[6]) allows Datacomp to analyze all input, responding immediately to changes in data compressibility.

III. Datacomp Design

A. Mechanism

Datacomp's mechanism transparently performs compression on behalf of the application via a user-space library providing `dcwrite()` and `dcread()`, function calls which wrap the familiar POSIX `send()` and `recv()` network calls (in their "blocking" mode). When `dcwrite()` is called, Datacomp chooses and executes a compression method on some amount of input, flushing the compressor to produce all compressed output, annotates the output with decompression instructions, and `send()`s the data. Like previous systems[11], [24], Datacomp uses separate compression and sending threads to pipeline the process. To ensure the integrity of the compressed output, Datacomp blocks until the entire compressed payload is transmitted. Datacomp then returns to the caller the number of bytes that were *effectively* sent (not the number of payload bytes actually sent). For example, suppose `dcwrite()` is called with 47KB of input and Datacomp chooses to compress only 32KB, which compresses to 20KB. Datacomp will `send()` 20KB but tell the caller that 32KB was sent so that the caller can re-send the remaining 15KB of input if desired.

Data sent by Datacomp is encapsulated so that the receiving side can properly decompress it. Datacomp payloads are prepended by a header of four four-byte fields: a magic number indicating the start of a Datacomp Frame (DCF), the length of the uncompressed payload, the algorithm and strength used to compress the input (two bytes each) and the length of the compressed payload (which forms the remainder of the DCF).

B. Methods

Datacomp supports arbitrary compressors using wrappers. The wrapper API takes four parameters: the algorithm family (e.g., zlib), the strength level (if applicable), how much data to compress and how many threads to use. This work used five compressor families providing a broad range of methods: "null" copies the data (but can vary size and thread count); LZO[17], a compressor that sacrifices compression ratio for speed; zlib[1], a ubiquitous and well-balanced compressor; bzip2[23], a stronger compressor using the Burrows-Wheeler Transform[26]; and xz[21], an aggressive and resource-hungry compressor. These libraries do not support parallelism natively, but Datacomp can parallelize them internally. Rather than supporting every strength level (which are not always very different), we supported three levels for each wrapper. For zlib and xz, these levels correspond to the strengths 1, 6 and 9. For LZO and bzip2, the "strength" parameter respectively selects algorithm variants and adjusts memory use.

Like other AC systems, Datacomp consumes data in fixed size chunks to simplify modeling. However, unlike previous work, Datacomp adaptively selects from five different quanta: 32KB, 64KB, 128KB, 256KB and 512KB. Datacomp also compresses "odd" sizes; a 40KB input can be compressed but will be modeled as a 64KB chunk or two 32KB chunks. Also, unlike some works, Datacomp will consider compressing inputs as small as 1KB. Payloads smaller than 1KB are left uncompressed because most algorithms have a practical limit in the hundreds of bytes below which data will not compress.

C. Monitors

Datacomp monitors four environmental properties: CPU utilization, CPU frequency, data type (i.e., compressibility) and available bandwidth. Datacomp obtains CPU load and current frequency from the `/proc` filesystem in Linux. To estimate the available bandwidth (ABW), Datacomp uses `ioctl` calls (present in Linux and BSD) that report the amount of data in the kernel send buffer for the socket. By tracking the sizes and times of buffer additions and using the `ioctl` call to measure drain rates, Datacomp estimates the rate at which the kernel is actually sending the socket's data.

1) Data Type Prediction: Datacomp does not estimate data properties based on context (e.g., file extension or recent results) or wasteful sample compression. Instead, it uses an efficient technique for compressibility estimation that we call "bytecounting" (BC). Examining the distribution of unique bytes in the input, BC returns a integral value that can be used as a compressibility estimate. Informally, the BC is the number of bytes in an input that appear at least as often as they would if every byte was equally represented. Formally, the BC is the number of unique bytes that appear in the input at least $threshold$ times, where $threshold = input\_length/256$.

A BC of 1 indicates that virtually the entire file is made up of one byte and thus should be highly compressible (no other bytes appeared $threshold$ times). A BC of 127 indicates that half of all possible bytes appeared "frequently." Natural language, sparse files and data with many repetitions have low byte counts and tend to be relatively compressible. Conversely, a high BC suggests "binary" data since text is limited to a smaller character set. Not only is binary data less likely to be compressible than text, but because $threshold$ is based on $input\_length$, a high BC also implies a relatively uniform distribution of bytes, which can enable incompressible data.

Level	Load	Freq.	ABW	BC
0	1-33	1-25		>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-200Mbit/s	1-33
4		95-100	20Mbit-200Mbit/s	
5			>200Mbit/s	

TABLE I
QUANTIZATION LEVELS. ALL COMBINATIONS ARE ALLOWED.

In a very loose sense, BC can be thought of as an entropy-*like* calculation. But, rather than measuring the information content of data, it is designed to efficiently predict compressor performance on real-world data. Calculating BC requires no heap allocations or logarithms and uses a single integer division operation – to compute $threshold$. Due to these and other optimizations, BC is roughly twice as fast as LZO on a per-byte basis. BC can also be used in conjunction with sampling to further increase prediction throughput. The low overhead of BC enables Datacomp to analyze all input, allowing it to quickly respond to changes in the data.

D. Models

Datacomp's model is based on the notion that the performance of every well-defined method-opportunity combination is constant (see Section II-B). If we can learn the best method for every opportunity, choosing the best method becomes trivial. Unfortunately, *perfectly* determining the current opportunity is unrealistic due to the number and the difficulty of identifying *all* significant factors. Instead, Datacomp approximates opportunities by monitoring a small number of factors that are known to be highly significant. However, monitoring a small number of factors at high resolution can become combinatorically overwhelming. Combining 100 levels of CPU load, 100 levels of CPU frequency, 127 levels for bytecount and 1,000 levels of bandwidth results in almost 1.3 billion unique opportunities. Datacomp copes with this complexity by quantizing the monitored values, reducing the number of opportunities recognized to approximately 400.

Quantization makes it feasible for Datacomp to track the average performance of every event type. When Datacomp needs to choose a method, it finds the results corresponding to the current opportunity and chooses the method with the best average EOR. After using the method, it updates the event record with the latest results.

Table I shows the quantization levels and their ranges for Datacomp's monitors. CPU load is roughly divided into thirds, with a fourth level reserved for 100% CPU load. Frequency is divided roughly into fourths, with a fifth level reserved for

95-100% of the maximum CPU frequency. We divided these ranges relatively evenly to avoid basing divisions on intuition or specific experiments. The exception to this is the special "top tier" bin for CPU load and frequency, which we added because maximum load and frequency states have distinct performance ramifications. Levels are independent; e.g., one unique opportunity is represented by load level 1, frequency 3, ABW 4 and bytecount 2.

Because data consisting of a single repeated byte has a bytecount of 1 and randomly generated (and thus incompressible) data has a bytecount near 127, we divided bytecount values below 100 into three roughly equal buckets with one bucket for bytecounts above 100. The quantization levels for ABW do not divide the range of bandwidths equally. Instead, they are similar to common real world network bandwidths: 1-768kbit/s for slow cellular connections; 768kbit/s-2Mbit/s for fast cellular or slow WiFi connections; 2Mbit/s-20Mbit/s for faster WiFi or slow LANs; 20Mbit/s-200Mbit/s for common LAN performance; and 200Mbit/s and above for gigabit LANs.

Quantization creates a tension between complexity and fidelity. Fewer quantization levels simplify the problem space by producing fewer discrete combinations, but this loss of resolution reduces accuracy because dissimilar data points are more likely to be aggregated in the same bin. On the other hand, while a greater number of levels (e.g., one-megabit ABW divisions) might result in better accuracy, it would definitely increase memory use and required training time.

The database for the model is a set of fixed-size memory-mapped tables for each opportunity composed of event records tracking the EOR for each method. This enables persistence and direct in-memory offset-based access without requiring read or write system calls. It also enables database persistence. Each record is composed of three 32-bit unsigned integers: the $average\_EOR$ (cached to avoid recalculation), the $count$ of events and the sum of the $count$ previous EOR values. When first initialized, the model contains no information, and will choose any method that has not been used at least twice until all methods have been used twice in the given opportunity. To facilitate adaptation and avoid overflows, Datacomp subtracts the current $average\_EOR$ from sum before adding the new result to sum if $count$ has reached 20.

One risk with this approach is that atypical events could skew the model. To help avoid this, the model occasionally artificially boosts a random record by increasing its $average\_EOR$ by 10% – without changing its sum or $count$. If the boosted average EOR causes the method to be used, a new $average\_EOR$ will be computed incorporating the new performance data.

Finally, thousands of data points generated during the design phase suggested that the vast majority of data with a bytecount of 100 or greater are not very compressible and thus are best handled using NC. However, "learning" this would require a long training phase on incompressible data. Based on the data, we added a shortcut so that regardless of the environment, Datacomp sends the next 256KB of data uncompressed when the bytecount is over 100. We felt this was an acceptable compromise because bytecounting (the expensive step in adaptation) is still performed every 256KB and this training shortcut could easily be reversed.

IV. Evaluation

Datacomp's primary goal is to improve time and energy efficiency by increasing effective throughput in realistic tasks without external assistance or significant hard-coded information. Additionally, rather than simply being profitable within these constraints, our secondary goals are that Datacomp will not only outperform any static compression method in the long run but will also approach optimal performance. We can easily measure relative improvement versus any static strategy, but evaluating the optimality of AC decisions requires knowing how much room for improvement exists. Achieving these goals for general-purpose systems is an added challenge, because it requires an evaluation that is representative of real-world workloads, which are numerous and diverse.

A. Environments

The primary experimental environment for this project was a private gigabit LAN and two personal computers, an Intel i7-920-based sender and an Intel i7-820-based receiver. Hyperthreading was disabled, resulting in four CPU cores being available for use. Along with their age, this made the test machines similar in computing power to modern laptops. We used dummynet[4] on the receiver to control bandwidth. We performed tests at four bandwidths: 1Mbit/s, 6Mbit/s, 100Mbit/s and 1Gbit/s. To adjust system load, we created a script, `bzip2urandom` (B2U), which reads data from `/dev/urandom`, compresses it, and writes it to `/dev/null`. Because this is such a computation-heavy task, each thread virtually monopolizes a single CPU core. B2U also performs a substantial amount of I/O, creating workload for the kernel as well. We modulated load by performing tests with zero, two and four B2U processes.

B. Data

To represent a broad range of "typical user data," we collected nine different types of data (called "pools") in three categories: *controlled*, *uncontrolled*, and *synthetic*. Controlled data comes from well-known, readily available sources and was collected in such a way that similar input should be readily available to others. It includes the executables (434MB) from an Ubuntu installation (called *binaries*) and two large personal mailboxes (called *mail*) with attachments (inline and Base64 encoded[12]) in the popular one-message-per-file Maildir format[3] (488MB). Controlled data also includes data from three web sites, collected using Chaosreader[10] to extract the components of web pages captured using Wireshark[5] or tcpdump[25] while surfing with encryption and compression disabled. *Wikipedia* data has a large text to graphics ratio relative to other sites, so it is fairly compressible. *Facebook* has a greater proportion of images, and is less compressible in comparison. *YouTube* data is primarily lossy-compressed video, so it is barely compressible. In all cases, a certain

number of megabytes was collected using a deterministic process: for Wikipedia, random featured articles were loaded (111MB); for Facebook, a user's "news feed" was consumed (154MB); for YouTube, the most popular videos in several top categories were viewed (210MB).

While the controlled classes consisted of popular and widely-available data, they were nevertheless selected by us. To include data from "real users," volunteers provided us with two types of "uncontrolled" data. First, web traffic was captured by five volunteers using the mechanism just described (*User Web*, 317MB). Volunteers were discouraged from using Facebook (which was already collected) or performing sensitive tasks (e.g., banking). Second, four volunteers, all heavy computer users from a variety of professions, provided virtually the entire contents of their home directories. From the files of each user, which primarily contained a mixture of documents and multimedia, 500MB of files were drawn (*User Files*, 2GB)[22]. Finally, "synthetic" data includes 100 1MB chunks of worst-case data (*random*) drawn from `/dev/urandom` and best-case data (*zero*), which is composed of one repeated byte.

The data resources in each pool were left in their original sizes rather than concatenating similar files into a large corpus. This ensured that Datacomp could not artificially benefit from the concatenation of many small or redundant inputs into a larger and potentially more compressible input. Because Datacomp's model uses a learning mechanism, each pool was divided into training and testing sets. These sets were then sampled in such a way as to mimic eavesdropping on a one-megabyte transfer of random files from that pool. These 1MB "file sets" were numbered and saved so that tests on the same input could be performed in different environments.

C. Experiment Design

`drcp` is a Datacomp-enabled remote copy utility. Where `drcp` would normally use `send()` and `recv()`, it instead uses Datacomp's `dcwrite()` and `dcsend()`. While transferring the training file sets with `drcp`, we modulated both the CPU load and ABW until all methods had been tried at least twice in each opportunity and performance stabilized. CPU frequency was dynamically controlled by the kernel.

Next, we made the model read-only. While in normal use the model would continually learn (i.e., there is normally no distinct "training phase"), we wanted to make sure that Datacomp would not unfairly learn how to compress the test data during encounters with the same file set in different environments. In the testing phase, we timed 25-50 `drcp` transfers of each type of test data within each combination of CPU load and ABW limit. Datacomp is fully capable of decompressing input, however, to focus on compress-and-send performance, the receiver discarded input.

Since most real-world applications use one compression method (or none), we wanted to quantify the difference between Datacomp and current approaches. To do this, we ran tests in matching environments with `drcp` in a "static mode" where it used one of NC, LZO or zlib with 512KB chunks and one compression thread. Apart from consuming input in 512KB chunks, no bytecounting, monitoring, history model or any other explicit AC overhead was paid during the static tests.

D. Comptool and Energy Measurement

The ideal performance and strategy for a given workload is generally unknown. Thus, while we can show Datacomp's ability to improve efficiency by comparing it to static strategies, we cannot – with these tools alone – quantify the optimality (or error) of Datacomp's choices and performance. To provide a best-case scenario against which to compare Datacomp, we built Comptool – an Adaptive Compression Oracle. Comptool tries and measures by brute force every combination of method and opportunity for a given input and, using a greedy strategy, constructs an event sequence optimizing some criteria, such as runtime or energy.

Comptool measures time using standard approaches. While it requires special hardware, Comptool can also use LEAP[7], a technology that captures high-resolution time-synchronized component-level energy consumption information. As with time, LEAP allowed us to determine the compression strategy resulting in the lowest energy consumption. By comparing the best strategies for a variety of workloads, we determined that Comptool's ideal energy strategy saved only 0.563% more energy than the ideal time strategy – less than the $\sim$1% overhead incurred by LEAP. Because of this (and because LEAP requires specialized hardware), we used "best time" as a very good approximation for the "best energy" strategy.

Comptool cannot be an effective AC mechanism due to its extreme cost, but it can provide a best-case performance estimate free of adaptation costs. Comptool is written in Python and is completely separate from Datacomp. Certain differences mean that Datacomp and Comptool's absolute performance in the same circumstances can differ slightly. (Comptool does not parallelize compression and its bandwidth throttling mechanism is only virtual.) However, by identifying the baseline performance of Datacomp and Comptool when using null compression, we can calculate and compare the relative difference in performance when using one of the dynamic or static strategies. Looking at these differences as improvement percentages allows us to perform a side-by-side comparison of the potential solutions.

V. Results

The following charts show the relative mean difference in run time between the specified strategy and NC as the bandwidth increases from 1Mbit/s to 1Gbit/s, expressed as a percentage. In other words, $y = 0$ represents the result achieved by sending the data without compressing it, $y = 50$ means that the mean run time of the given method was half that of no compression and $y = -50$ indicates that run time was *increased* by half (i.e., 1.5 times longer). A y-value of 100 is impossible because it would mean that the method runtime was 0, while a y-value of -100 indicates that run time doubled. In addition to null ($y = 0$), CT indicates the

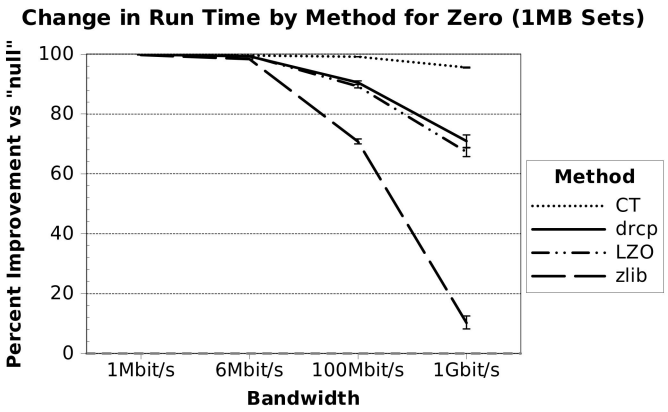

Fig. 1. Improvement (vs. no compression) by strategy for *zero* data.

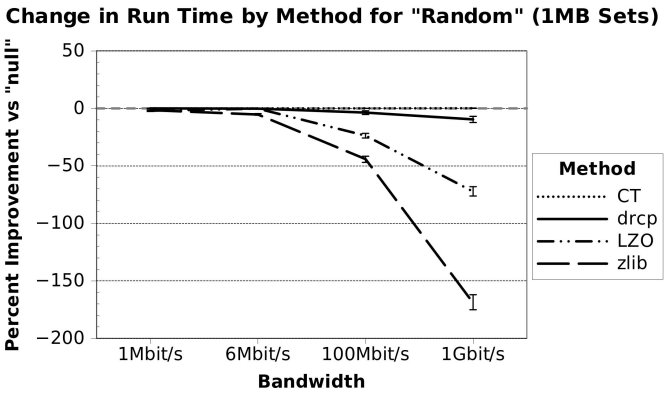

Fig. 2. Improvement (vs. no compression) by strategy for *random* data.

ideal improvement derived by Comptool, `drcp` indicates the improvement of Datacomp's throughput model, and LZO and zlib indicate the improvement gained by the respective static strategy. Error bars were computed using the BCa bootstrap with 20,000 samples and show the 95% confidence interval. A rough estimate for the run time of NC is $sample\_size/ABW$; for example, at 1Mbit/s, a 1MB transfer takes on the order of 8s, while at 1Gbit/s it takes approximately 0.008s.[1]

A. Zero and Random Data

Shown in Figures 1 and 2, *Zero* and *random* are best- and worst-case types useful as bookends for the range of improvements possible through compression. *Zero*'s results should have the highest possible gains. At 1Mbit/s, all non-null methods reduce run-time by over 99%, reducing runtime by almost three orders of magnitude. At 6Mbit/s, while all methods are well above 90%, we begin to see that zlib, LZO and `drcp` lose ground to CT. LZO and `drcp`'s closeness suggests that Datacomp is largely using LZO but that it is just not as efficient as CT's idealized behavior. (We did not track Datacomp's specific choices during testing in order to avoid hurting performance.) At 100Mbit/s, while CT still manages to improve by 99%, `drcp` and LZO fall to ∼90% and zlib to ∼71%. At 1Gbit/s, all methods are less profitable; zlib improves by only 10%, LZO and `drcp` improve by 67-71% and CT falls to 96%.

These results establish several trends that recur throughout the results. First, most methods are generally equivalent at 1 and 6Mbit/s. Next, although it can still be profitable, zlib is rarely the best choice at 100Mbit/s and is never the best choice at 1Gbit/s; instead, it is often worse than NC at these ABWs. Third, Comptool always performs extremely well and often squeezes out statistically significant benefits at 1Gbit/s. However, it is important to remember that Comptool's results *only include compression and transmission time* – no AC overhead is included. Finally, `drcp` and LZO are sometimes able to perform well at 100Mbit/s, but are unable to match Comptool at 1Gbit/s. Nevertheless, they can still improve performance over NC if the data is sufficiently compressible.

[1]Variance is introduced by implementation issues (e.g., dummynet throttling and software or networking overhead).

Figure 2 shows the results for *random* data. Because the data is incompressible, we expect no improvement from any technique – breaking even is the best case. CT does this at all bandwidths, showing that it correctly chooses NC. In contrast, even at 1Mbit/s, `drcp`, LZO and zlib are respectively 0.008%, 2% and 1.5% slower than NC. LZO and zlib are penalized because they perform compression even though there is no possible benefit. `drcp` is penalized 0.008% at 1Mbit/s because, even though it correctly chooses NC, it must still pay adaptation costs, such as bytecounting.

The value of compression decreases as ABW increases (per Equation 1) because all compressors require CPU time and higher ABW means less time to compress. This effect is most visible for incompressible data, which shows penalties even at low bandwidths because there is no benefit to offset any costs. However, this occurs even for compressible data, because even the fastest compressor will eventually become a bottleneck if bandwidth increases enough. While `drcp` does not compress *random* data, it still pays the cost of bytecounting. This results in a loss of ∼4% at 100Mbit/s and ∼10% at 1Gbit/s.[2] Nevertheless, `drcp` still outperforms static strategies at 1Gbit/s (where LZO and zlib respectively cost ∼72% and ∼169%) because bytecounting prevents `drcp` from wasting even more resources.

B. Wikipedia, Facebook and YouTube Data

Results for *Wikipedia* (*WP*) data are shown in Figure 3, where at 1Mbit/s and 6Mbit/s, CT, `drcp`, LZO and zlib improve performance by between 60 and 70%. For *WP* data at 100Mbit/s, CT, `drcp`, and LZO achieve around 50%, while zlib is clearly suboptimal, improving throughput by only 16%. At 1Gbit/s, CT predicts potential improvements of ∼28%, but `drcp` and LZO are only able to break even. At the same time, zlib more than doubles the run time of NC (∼-134%).

Facebook performance (Figure 4) is similar, but with less overall improvement owing to less compressible data. At 1Mbit/s and 6Mbit/s, all methods are statistically similar, improving by 33-40%. At 100Mbit/s, CT, `drcp` and LZO reduce run time by 16-27%, while zlib *increases* runtime by

[2]Database access and other AC costs are much smaller than the bytecounting cost.

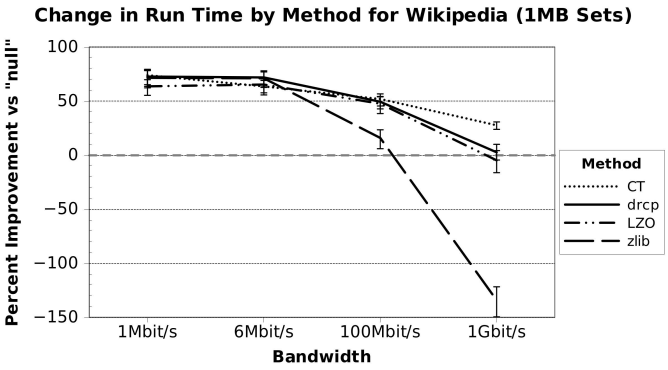

Fig. 3. Improvement (vs. no compression) by strategy for *Wikipedia* data.

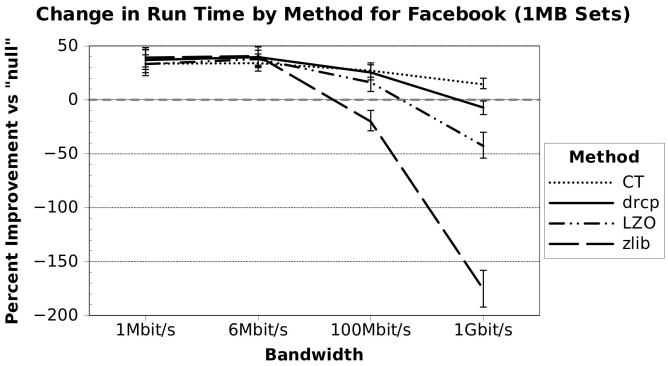

Fig. 4. Improvement (vs. no compression) by strategy for *Facebook* data.

20%. At 1Gbit/s, CT estimates a potential 14.3% improvement. While `drcp` costs 7% (due to bytecounting costs and *FB*'s lower compressibility), this penalty is much less than LZO (43%) and zlib (176%).

Results for YouTube (*YT*) data are shown in Figure 5. Over all bandwidths, CT's analysis estimates a potential improvement of 1.3-2.4% because, unlike *random*, *YT* is *slightly* compressible. As a result, at 1Mbit/s and 6Mbit/s, `drcp`, LZO and zlib manage to break even with NC. However, at 100Mbit/s, `drcp` still manages to break even, but LZO and

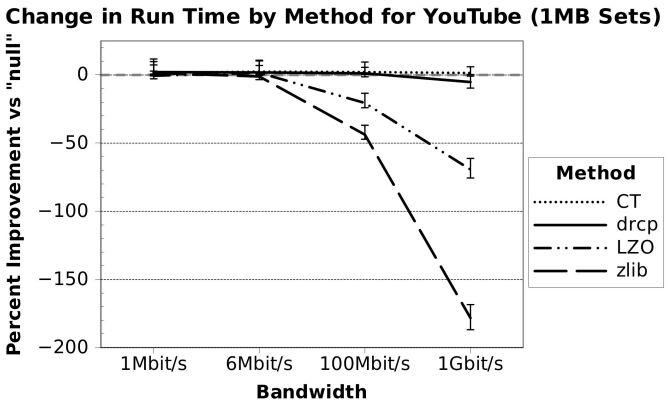

Fig. 5. Improvement (vs. no compression) by strategy for *YouTube* data.

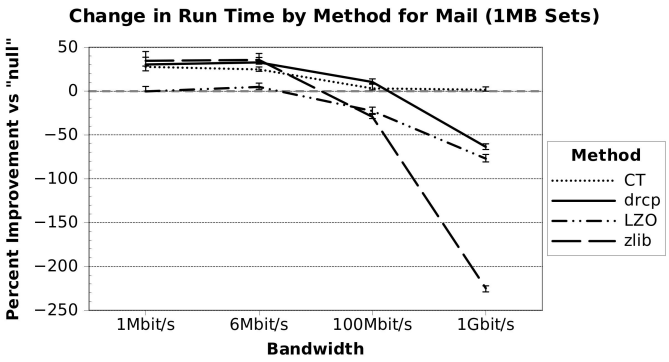

Fig. 6. Improvement (vs. no compression) by strategy for *mail* data.

zlib lose ∼21% and ∼44%. At 1Gbit/s, `drcp` loses 5% to bytecounting – much better than LZO (-69%) or zlib (-178%).

Looking at these three types of data, we see that the best static choice (NC, LZO or zlib) depends on both the data and bandwidth – no single method is always ideal, and some choices are very costly. In contrast, Datacomp's dynamic performance is statistically similar to Comptool's analytically derived result for 1Mbit/s through 100Mbit/s. While Datacomp can cause a small loss at 1Gbit/s, there are potential solutions to this issue and it nevertheless outperforms static compression methods like zlib and LZO (See Section VI).

C. Mail and Binaries

Needed at a time when not all I/O channels were "8-bit clean," Base64[12] encoding represents binary data using 64 printable characters, such as [A-Za-z0-9+/]. Because Base64 encodes a sequence of three 8-bit bytes of input as four 7-bit bytes (which are stored as 8-bit bytes in modern systems), it expands data by approximately 25% without adding information and is thus always compressible, even if the original input was random. However, compression performance on Base64 data varies significantly by method. In particular, while CT, `drcp` and zlib all manage to improve performance by 28-35% for *mail* at 1Mbit/s, LZO only *breaks even with NC*. At 100Mbit/s, even though `drcp` manages to improve performance by ∼10%, zlib and LZO lose 29% and 23%.[3] Similarly, although binaries are also generally quite compressible, zlib's output rate is much lower for binaries than for other data types. As a result, even though under zlib the entire *binaries* pool has a CR of 0.41, zlib loses 246% at 1Gbit/s – nearly 70% worse than its 1Gbit/s performance on *random* data.

These results are interesting for a number of reasons. They show that even though LZO is fast in general, it is not always a "safe" choice (or profitable) for compressible data – even at or below 100Mbit/s. Additionally, sometimes a particular compressor is unable to effectively compress data although it is compressible by other methods; where `gzip -6` achieves a CR of 0.68 on a concatenation of *mail* input, LZO only achieves 0.92. The zlib results for binaries show that while random data may be the worst case from a compressibility perspective, it is not necessarily the worst

[3] `drcp`'s results at this bandwidth are likely due to parallelization.

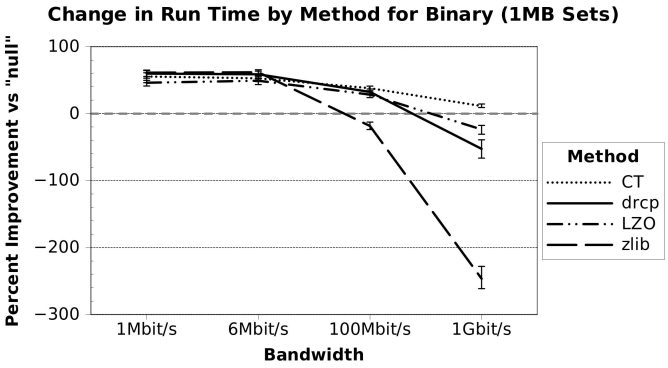

Fig. 7. Improvement (vs. no compression) by strategy for *binary* data.

case from a throughput perspective, a fact that is critical for AC choices. Comparing LZO and zlib's results for *mail*, *Facebook*, *YouTube* and *Wikipedia*, we also see that LZO's improvement percentage is not predictable based on zlib's performance alone. For instance, while zlib's improvement at 1Mbit/s for *Facebook* (39%) and *mail* (35%) are similar, LZO's results (33% for *Facebook* and 0% for *mail*) are very different. Also, while LZO typically becomes more profitable than zlib somewhere between 6Mbit/s and 100Mbit/s, the ABW and improvement percentage at which this occurs varies significantly by input type. These conclusions all highlight the importance of considering data type for AC decisions.

While `drcp`'s performance on *mail* between 1Mbit/s and 100Mbit/s is equivalent to Comptool's analytic "best," it makes a significant misprediction at 1Gbit/s, resulting in a loss of nearly 64%. This is considerably worse than the expected bytecounting loss at 1Gbit/s of ∼10% (Figure 2), so we know that `drcp`'s model is erroneously choosing to compress. Similarly, while `drcp` performs well on *binaries* between 1Mbit/s and 100Mbit/s it makes a significant misprediction at 1Gbit/s, losing nearly 53%. Fundamentally, these errors are caused by *Opportunity Aliasing* in Datacomp's learning model.

Opportunity Aliasing (OA) occurs when significantly different opportunities (see Section II-B) are treated as a single group during training or use, as through quantization. For example, in this work, bytecount is divided into four levels to reduce model complexity, resulting in a range of about 33 bytecount values per level (Table I). Aliasing can also occur prior to quantization if monitor resolution is too low. For example, it is possible to construct two files with identical bytecounts but different CRs. Like adaptation costs (e.g., bytecounting), the deleterious effects of OA increase with bandwidth for two related reasons: fewer compression strategies are able to improve performance and mistakes become more costly.

Unfortunately, some amount of Opportunity Aliasing is unavoidable when estimates, rounding, statistical summaries or quantization are used. Dynamic learning mechanisms are one way to mitigate the effects of aliasing. Had `drcp` been allowed to learn during testing, the pathological combinations of LZO+*mail* and zlib+*binaries* should have quickly reduced their average EOR as recorded in the model, resulting in more effective methods being used. Another method to minimize aliasing might be to select and adjust quantization levels dynamically, which could result in outlier types being treated individually. Finally, aliasing, like bytecounting cost, is most visible at higher bandwidths, because profit margins are thinnest and mistakes are more costly. These errors would not have occurred if `drcp` did not compress if the ABW was above some local limit (see Section VI).

D. Other Tests

Performance was similar for the data pairs *User web* and *Facebook*, and *User Files* and *YouTube*. This is presumably due to similar compressibility; *User Files* contains a large amount of compressed data relative to uncompressed data (like *YouTube*) and modern websites include a large amount of compressible text, including markup, stylesheets and scripting (like *Facebook*). While not shown due to space constraints, `drcp` coped well with varying levels of CPU load generated by B2U [22]. However, B2U not only heavily loaded the CPU but also decreased I/O throughput (likely due to the large number of I/O requests). For example, without B2U `drcp` lost 64% on *mail* data at 1Gbit/s, but broke even in that scenario with four B2U processes. This could be due to `drcp` correctly modeling the effect of CPU load, but throughput measured during the test was about four times lower with four B2U processes. Since lower ABW makes AC improvement easier, it is difficult to separate the effect of CPU contention generated by B2U from the reduction in overall throughput.

E. Future Work

Datacomp as described is a foundation for future development. Datacomp's lookup table model and quantization levels are somewhat ad hoc and naïve, chosen as a first approach due to their simplicity and efficiency. However, Datacomp's methods, monitors, models and methods are all changeable. Enhancements to its model, such as increasing the granularity of its quantization, a sensitivity study for quantization factors, dynamic quantization, or a more traditional machine learning approach could improve accuracy and robustness. If they can be accurate and fast enough, they may be able to improve performance and improve efficiency. Similarly, we may be able to improve Datacomp's compression choices and compressibility estimation by including compressors faster than LZO, such as Snappy[9] or by comparing bytecounting against optimized entropy-based estimation approaches.

Improvements to existing experiments are also planned, including a CPU contender that does not make I/O requests and potentially an I/O-only contender. The impact of receiver-side decompression is also important to explore. New types of experiments and environments are planned, including comparison with existing remote copy tools that incorporate compression and evaluation in mobile, embedded and/or sensor environments where conditions can change rapidly and efficiency is critical.

VI. Conclusion

Datacomp is an Adaptive Compression system that improves the throughput of network communication by choosing

and using the best compression methods for every opportunity. Datacomp is designed to be useful for devices like laptops and smart phones, so it performs its work locally, without precomputed models or remote support. At 1-100Mbit/s the performance of Datacomp's adaptive strategy is always either statistically similar or very close to the Comptool oracle's estimate. Our results also show that no single compression method is optimal; in some test, using a fixed method is either significantly suboptimal or costs more than not compressing. At 1Gbit/s, the current version of Datacomp generally fails to break even because any per-byte adaptation costs will *eventually* be unrecoverable as bandwidth increases. Potential solutions to this problem include disabling analysis and falling back to "no compression" at higher bandwidths.

Unusually poor performance by LZO on *mail* and by zlib on *binary* data shows that relationships between methods do not hold across all data. Datacomp does not assume relationships between methods, but it does assume that opportunities are well-defined and thus that method performance for an Opportunity will be consistent. Unfortunately, due to Opportunity Aliasing (OA) and decreasing profits at higher bandwidths, the current version makes costly mistakes for these data types in gigabit scenarios. This highlights the need for improved opportunity modeling. Dynamic approaches for adjusting the number and divisions of quantization levels or using multiple data characterization methods could improve the accuracy of each "bucket," reducing OA. Of course, more monitors and more complicated modeling could increase overhead.

Finally, while Datacomp performs well across a broad range of realistic types of data and bandwidths, more varied tests are necessary to show Datacomp's utility. Tests with dynamic data types and environmental conditions could show Datacomp's flexibility, and tests with Datacomp-enabled versions of real-world software would demonstrate feasibility. In addition, Datacomp may be able to improve the general efficiency of other distributed environments, such as office LANs, server farms, sensor networks or even cloud computing environments.

In summary, compared to any static strategy (including not compressing) Datacomp improves efficiency and performance in a wide range of environments on realistic data without requiring precomputed models or external support. At 100Mbit/s. where compression is often disabled to avoid degrading performance, Datacomp reduces transfer time of *Wikipedia* data by an average 33% more than zlib and 49% more than choosing not to compress. For *mail*, Datacomp improves runtime by 10% at 100Mbit/s while LZO and zlib degrade it by 23% and 29%. For worst-case data at 100Mbit/s, Datacomp loses only 4% while LZO and zlib lose 24% and 44%. While there is room to improve, these results show that Datacomp has significant potential for improving the efficiency of real-world computer systems.

VII. Acknowledgments

This work was supported under NSF grant CNS-1116898. We sincerely thank the anonymous reviewers for their thoughtful and constructive criticism.

References

[1] Adler, M., and Roelofs, G. zlib home site. http://www.zlib.net. Accessed 07/14/2014.

[2] Barr, K., and Asanovic, K. Energy-aware lossless data compression. *ACM Trans. Comput. Syst.* (2006).

[3] Bernstein, D. J. Using maildir format. http://cr.yp.to/proto/maildir.html. Accessed 06/07/2013.

[4] Carbone, M., and Rizzo, L. Dummynet revisited. *ACM SIGCOMM Computer Communication Review* (2010).

[5] Combs, G. Wireshark – go deep. http://www.wireshark.org/. Accessed 06/07/2013.

[6] Culhane, W. Statistical measures as predictors of compression savings. *The Ohio State University (Undergraduate Honors Thesis)* (2008).

[7] Google, Inc. Data compression proxy - Google chrome mobile. https://developers.google.com/chrome/mobile/docs/data-compression. Accessed 06/07/2013.

[8] Google, Inc. Snappy by google. http://google.github.io/snappy/. Accessed 04/28/2016.

[9] Gregg, B. Chaosreader. http://chaosreader.sourceforge.net/. Accessed 06/07/2013.

[10] Hovestadt, M., Kao, O., Kliem, A., and Warneke, D. Evaluating adaptive compression to mitigate the effects of shared i/o in clouds. *IEEE Internation Parallel & Distributed Processing Symposium* (2011).

[11] Jeannot, E. Improving middleware performance with AdOC: an adaptive online compression library for data compression. *19th IEEE International Parallel and Distributed Processing Symposium* (2005).

[12] Josefsson, S. RFC 4648 - the base16, base32, and base64 data encodings. http://tools.ietf.org/html/rfc4648. Accessed 07/15/2014.

[13] Knutsson, B., and Bjorkman, M. Adaptive end-to-end compression for variable-bandwidth communication. *Computer Networks* (1999).

[14] Motgi, N., and Mukherjee, A. Network conscious text compression system (NCTCSys). *Proceedings of the IEEE International Conference on Information Technology: Coding and Computing* (2001).

[15] Nicolae, B. On the benefits of transparent compression for cost-effective cloud data storage. *Transactions on large-scale data-and-knowledge-centered systems III* (2011).

[16] Oberhumer, M. F. X. J. oberhumer.com: LZO real-time data compression library. http://www.oberhumer.com/opensource/lzo/. Accessed 07/07/2013.

[17] Opera Software. Opera help: Opera turbo. http://help.opera.com/Linux/10.60/en/turbo.html. Accessed 06/07/2013.

[18] Pandya, G. Full disclosure: Nokia phone forcing traffic through proxy (mailing list). http://seclists.org/fulldisclosure/2012/Dec/95. Accessed 06/07/2013.

[19] Park, K. W., and Park, K. H. ACCENT: Cognitive cryptography plugged compression for ssl/tls-based cloud computing. *ACM Transactions on Internet Technology* (2011).

[20] Pavlov, I. LZMA SDK (Software Development Kit). http://7-zip.org/sdk.html. Accessed 07/07/2013.

[21] Peterson, P. A. H. *Datacomp: Locally Independent Adaptive Compression for Real-World Systems*. PhD thesis, The University of California, Los Angeles, 2013.

[22] Seward, J. bzip2: Home. http://www.bzip.org/. Accessed 06/07/2013.

[23] Singh, D., Peterson, P., Reiher, P., and Kaiser, W. J. The Atom LEAP Platform For Energy-Efficient Embedded Computing: Architecture, Operation, and System Implementation, 2010.

[24] Sucu, S., and Krintz, C. Adaptive on-the-fly compression. *IEEE Transactions on Parallel and Distributed Systems* (2006).

[25] team, T. T. Tcpdump/libpcap public repository. http://www.tcpdump.org/. Accessed 04/28/2016.

[26] Wheeler, M., and Burrows, D. A block-sorting lossless data compression algorithm. *DEC Systems Research Center* (1994).

[27] Wiseman, Y., Schwan, K., and Widener, P. Efficient end to end data exchange using configurable compression. *ACM SIGOPS Operating Systems Review* (2005).

[28] Wolski, R. Dynamically forecasting network performance using the network weather service. *Cluster Computing* (1998).

[29] Xiao, Y., Siekkinen, M., and Yla-Jaaski, A. Framework for energy-aware lossless compression in mobile services: the case for e-mail. *IEEE International Conference on Communications* (2010).