To Zip or not to Zip: Effective Resource Usage for Real-Time Compression[*]

Danny Harnik, Ronen Kat, Oded Margalit, Dmitry Sotnikov, Avishay Traeger
IBM Research–Haifa
{dannyh,ronenkat,odedm,dmitrys,avishay}@il.ibm.com

Abstract

Real-time compression for primary storage is quickly becoming widespread as data continues to grow exponentially, but adding compression on the data path consumes scarce CPU and memory resources on the storage system. Our work aims to mitigate this cost by introducing methods to quickly and accurately identify the data that will yield significant space savings when compressed.

The first level of filtering that we employ is at the data set level (e.g., volume or file system), where we estimate the overall compressibility of the data at rest. According to the outcome, we may choose to enable or disable compression for the entire data set, or to employ a second level of finer-grained filtering. The second filtering scheme examines data being written to the storage system in an online manner and determines its compressibility.

The first-level filtering runs in mere minutes while providing mathematically proven guarantees on its estimates. In addition to aiding in selecting which volumes to compress, it has been released as a public tool, allowing potential customers to determine the effectiveness of compression on their data and to aid in capacity planning. The second-level filtering has shown significant CPU savings (up to 35%) while maintaining compression savings (within 2%).

1 Introduction

Data continues to grow at an exponential rate, outpacing the growth of storage capacity [9]. Data is also growing faster than IT budgets are, making it difficult to purchase and maintain the disk drives required to store necessary data [8]. Historically, compression has been widely used in tape systems, desktop storage, WAN optimization, and within applications such as databases. In the past few years, deduplication and compression have been widely adopted in backup and archive storage. More recently, enterprise storage solutions have been adopting compression for primary storage.

In practice, primary storage compression comes in three main flavors: real-time or inline compression [2, 15, 21–23], offline compression [6], and a mix of real-time and offline [18]. Our work focuses on real-time compression for block and file primary storage systems. However, its basic principles are general and can be applied to other compression use cases as well.

[*]This research was partially supported by the European Community's Seventh Framework Programme (FP7/2001-2013) under grant agreement number 257019 - VISION Cloud.

Compression does not come for free: it consumes system resources, mainly CPU time and memory. It may also increase latency for both write and read operations, where data must be compressed and decompressed, respectively. On the other hand, if compression reduces the data size substantially, performance can improve; disk I/O time is reduced, and the system's cache can hold more data. These benefits can neutralize the compression overheads, and in some cases overall performance may even improve substantially.

But what if the compression yields little or no benefit for a given data set? In such cases, not only are there no capacity savings (and associated cost savings), but also degraded performance and wasted resources. Worse yet, standard LZ type compression algorithms incur higher performance overheads when the data does *not* compress well. Figure 1 shows that with the standard *zlib* library [5, 7], one invests $3.5X$ (or more) as much time when compressing data that compresses to 90% of its original size as it does when compressing data that reduces to 20%. These observations indicate that it is advisable to avoid compressing what we refer to as "incompressible" data.

Our work addresses the problem of identifying incompressible data in an efficient manner, allowing systems to effectively utilize their limited resources: compressing only data that will yield benefits, and not compressing data that will cause the system to needlessly waste resources and incur unnecessary overheads. Our solution is comprised of two components: a *macro-scale* compression estimation for the data set, and a *micro-scale* compressibility test for individual write operations.

Our macro-scale estimation tool is an efficient *offline* method for accurately determining the overall compression ratio and distribution of compression ratios for a data set (e.g., volume or file system). It runs in a matter of minutes and provides tight statistical guarantees on its estimates. The output of the estimate can result in three main actions. First, if the estimate shows that the data is highly compressible overall, compression can be enabled with confidence that little-to-no resources will be wasted. Second, if the output indicates that compression will yield little benefit, compression can be disabled, allowing resources to be directed to more beneficial tasks. Third, if the data set contains data of varying compressibility which has the potential to yield significant savings, but where compressing all data would be wasteful, the micro-scale test can be enabled to safely enable compression on a finer granularity.

The micro-scale test examines data as it is written and for each write buffer, quickly determines the effectiveness of compressing it. This *online* test must be quick and accurate to maximize the benefits of compression. If it is not significantly faster than compression, performance will be similar to compressing all data. If it is not accurate, we may miss opportunities to compress data or waste resources on incompressible data. A common method to determine if a write buffer is compressible is to compress the first portion and see how well it compresses. We call this *prefix estimation*. If the prefix compresses well enough, the remainder of the buffer will then be compressed, and so there is no overhead because no extra work is done. However, the micro-scale test is most important when there is a significant amount of incompressible data, and here prefix estimation has a high overhead (recall that compression is most expensive for incompressible data). We have developed a method to complement prefix estimation that samples the data buffer and employs heuristics to determine its compressibility. For incompressible data, it has 2–6X less performance overhead, and overall incurs less capacity overhead due to missed compression opportunities.

Our contributions: The contributions of this paper are the designs and implementations of the macro- and micro-scale tests, along with thorough experimentation and analysis on real-world data.

The macro-scale test provides a quick and accurate estimate for which data sets to compress. It has been released to the public as a tool for use with real-time compression on the IBM Storwize V7000 and SAN Volume Controller [13]; it has been proven in the field to be an effective tool for customers to determine the effectiveness of compression on their data, the amount of storage to purchase, and which volumes to compress. It is also being used in a cloud storage prototype to selectively enable compression for large data transfers over the WAN. The results are mathematically proven to provide a well-defined degree of accuracy, so its estimates can be trusted.

The micro-scale test heuristics have proved critical in reducing resource consumption while maximizing compression for volumes containing a mix of compressible and incompressible data. The heuristic method managed to reduce CPU usage by up to 35% on a representative data set, providing a significant improvement to a storage system, which must effectively use its generally scarce resources to serve I/Os.

Paper organization: Section 2 provides technical and performance background on compression. We discuss the macro-scale estimation in Section 3 and the micro-scale in Section 4. Section 5 describes related work and Section 6 provides concluding remarks.

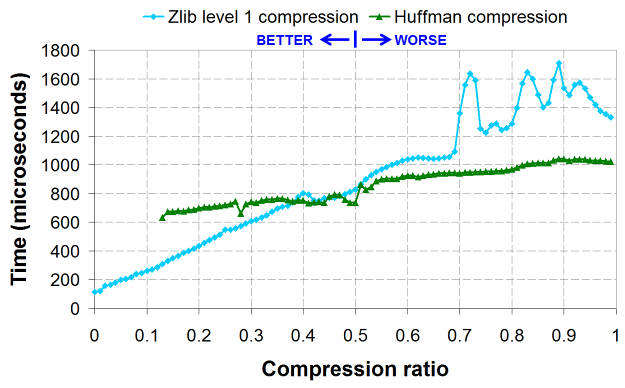


Figure 1: The average times to compress 32KB of data for each compression ratio, as collected by compressing the Benchmark *data set described in Table 3.*

2 Preliminaries

The vast majority of *lossless* compression algorithms are based either on Huffman encodings [12] or the Lempel-Ziv algorithms [24, 25]. Huffman codes make use of the fact that in most data not all symbols are equally popular, and encodes the most frequent symbols by the shortest bit representations. The Lempel-Ziv methods look for repeating strings in the data and store back-pointers to long repetitions, thus avoiding the need to repeat data and ultimately shortening the representation. Perhaps the most widely used method is the DEFLATE framework which combines the two aforementioned methodologies (namely a Huffman encoding on top of an LZ77 encoding). Our study is centered around this framework which is relevant to IBM's Real-time Compression [22], but it's general concepts are applicable for many other compression methods.

By *compression ratio*, we refer to the ratio between the size of the compressed data to the size of the original data. We present the compression ratio on a scale from 0 to 1, where compression savings are greater as the ratio is closer to zero. Alternatively we sometimes use percentage to present this ratio.

In our tests we use the popular zlib open source compression library [5, 7] that implements the DEFLATE algorithm. zlib implements a number of different *compression levels* – numbered between 0 and 9: 1 gives the best speed, while 9 attempts to give the best compression ratio, and 0 gives no compression at all (the input data is simply copied). In this work we use level 1 as it provides the fastest compression, which is an important factor for real-time compression, which is our focus. Our implementation disabled compression by using zlib level 0, and set the `HUFFMAN_ONLY` flag to disable the LZ77 algorithm and use only the Huffman coding.

Figure 1 shows the impact of varying compressibility on the CPU time (for zlib level 1 and Huffman coding on 32KB data blocks). Note that compression time for zlib level 1 varies linearly by more than a factor of 8, performing worst for data that does not compress well (what we refer to as *incompressible*). The run time of the Huffman-only coding option is also influenced linearly by compression ratio, and is faster than zlib level 1 for compression ratios higher than 0.5.

3 The Macro-scale Solution

The macro-scale solution is appropriate for scenarios that consider compression on a large scale; for example, an entire volume or file system of a storage system, or a large data transfer. The solution constitutes an efficient estimation mechanism for the aggregate compression ratio over the entire data set. The input to the procedure is a pointer to the data set (e.g., a physical volume, a file system, or a list of objects) and the output is an overall compression ratio, histogram, and an accuracy range.

We implemented our estimator with two basic interfaces. The first is a block device interface, which is tailored for a specific real-time compression product that compresses data stored on a block storage system. The second is a file and object storage interface. The input here is a list of files or objects and their sizes. In this model each object is compressed independently using a standard compression mechanism (we use zlib with compression level 1). This implementation can estimate compression for both a file system and a collection of objects.

Our solution possesses two key properties that make it particularly useful: speed and accuracy.
Speed: The test runs very fast, especially when contrasted with the time an exhaustive compression would take on a large volume (improvements of up to 300X over an exhaustive run are not uncommon). The run time depends on the underlying platform and storage speed, but is typically under two minutes, *regardless of the volume size*. Performance results are described in Section 3.3.
Accuracy: The estimation comes with a statistical accuracy guarantee of the following form: the probability that the estimation will be off by an additive error of A (the *accuracy*) is no more than $1 - C$ (the *confidence*). In a typical example, A could be 5% of the volume size, and the guarantee would be that the test would be off by more than A only once in 10 million tries (here $C = 10^{-7}$). This guarantee is backed up by a mathematical analysis (see Section 3.2).

3.1 Motivation and Use Cases

Our estimation tool has proved most useful in a variety of use cases which rely on its speed and accuracy.

Per-volume decisions: It is common to build a storage system of multiple volumes, with each volume serving a specific application. When configuring a system to include compression, one can run a quick per-volume estimation to decide whether or not to compress each volume. Determining when to compress is critical, as every system has limited resources, and so it is unwise to direct resources to compressing data where there is little or no benefit. In many cases, the volume will contain data with homogeneous compression characteristics, and compression can be enabled or disabled for the entire volume. Often, however, the volume will contain data with varying degrees of compressibility. The macro-scale test provides a histogram of compression ratios that clearly shows this case and allows enabling the micro-scale test described in Section 4 to compress only the portions of the volume where it is beneficial. New volumes with little data can initially be saved in uncompressed form and periodically tested with the estimation tool. Once the volume's characteristics are clear, a decision can be made.

Sizing and pre-sales: The high accuracy provided by the estimation allows determining what the expected size of a storage system would be if it has compression included. This is instrumental in two key junctures: 1) Evaluating the overall cost savings that can be achieved by integrating compression into a system—naturally, this provides a powerful selling tool for systems with compression, and 2) Deciding how many disks to buy for a system with compression. The more accurate the estimation, the more money customers can save by not purchasing extra disks to over-provision their systems. Indeed, our implementation is an official planning and pre-sales tool for two IBM storage systems that support real-time compression.

Large data transfers: Large data transfers between hosts with limited bandwidth are typical in cloud storage scenarios. For example, in large multi-site storage clouds, large amounts of data need to be transferred over a wide area network (WAN) when ingesting data from cloud users or when recovering from a site failure. Compressing data can reduce transfer time, but unless the compression provides enough benefit, it may actually be slower than transmitting the uncompressed data. By running our estimation method, one can quickly and accurately compute the benefits of compression. Our estimation in this case also takes the connection speed into account to determine the effectiveness of compression in improving transfer time. The macro-scale test generally completes in under two minutes, while the data transfers may take hours or even days to complete, and can be crucial in reducing transfer time as well as limiting resources used for compression. The resulting decision is similar to the case of per-volume decisions: enable or disable compression completely, or decide on smaller granularities using the micro-scale test.

3.2 The General Method

Our method is based on random sampling, within the following natural and general framework:

1. Choose m random locations in the data, where m is a predefined sample size (see Section 3.2.1).
2. For each location compute a compression ratio.
3. The overall estimation is an average of the m compression ratios.

While the above process is very simple and expected, the devil lies in the details. There are two main issues that must be addressed:

- How to choose the number of samples m?
- How to compute the local compression ratios for each sampled location?

Our method is complemented with an analytical proof of accuracy that answers both of the above questions. Our method is guided by the proof, which dictates valid sampling methods, as well as how many samples are required to achieve a desired level of accuracy.

3.2.1 Proof of Accuracy and Choosing the Sample Size

In a nutshell, we look at the total compression ratio as an average of the compression ratio of each byte in the data set. This is tricky, however, as bytes are not compressed independently and there is no meaning to the compression ratio of a single byte. Instead, we view the *contribution* of a single input byte to the output of the compressed stream as an average of the compression rate in its *locality* (we define locality in Section 3.2.2). Let $contribution_i$ denote the *contribution* of the i^{th} byte in the volume. Our goal is to define contribution so that the overall compression ratio could be presented in this form:

$$\text{Compression Ratio} = \frac{1}{n}\sum_{i=0}^{n-1} contribution_i$$

The benefit of this representation is that estimating an average over a large population is a well studied problem in statistics. Specifically, we know that the average of a random sample of the population is a good estimator and can bound the variance of this estimator as a function of the number of samples. We do this using the Hoeffding inequality [11] which relates between three parameters: *Accuracy*, *Confidence* and the sample size m. The analysis states that if

$$Confidence = 2e^{-2m*Accuracy^2}$$

Then

$$Prob(|Estimation - Ratio| > Accuracy) < Confidence$$

Accuracy	Confidence	Sample Size
0.05	10^{-7}	3363
0.02	10^{-7}	21015
0.01	10^{-7}	84057
0.05	10^{-3}	1521
0.05	10^{-6}	2902
0.05	10^{-9}	4284

Table 1: Sample size examples for different values of accuracy and confidence.

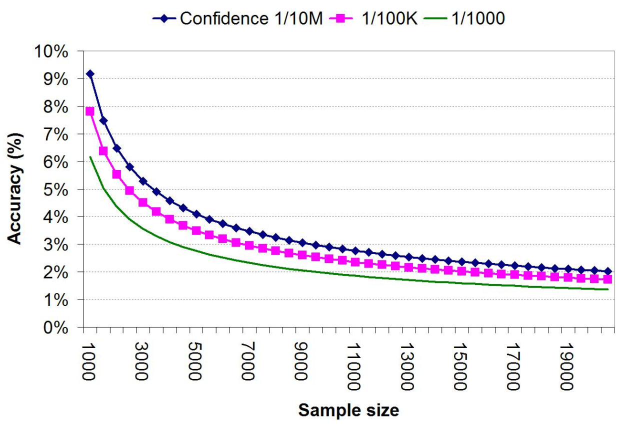


Figure 2: Accuracy as a function of sample size for 3 levels of confidence.

That is, the estimation will stray by more than *Accuracy* with probability at most *confidence*.

From this analysis we deduce the number of samples that should be used in the sampling mechanism as a function of the accuracy and confidence parameters, as stated in the following formula:

$$m \geq \frac{1}{2 \cdot Accuracy^2} \ln\left(\frac{2}{Confidence}\right)$$

Note that the confidence parameter appears inside a logarithm and does not influence the result greatly. In fact one can greatly improve the confidence with little effort (see example in Table 1). On the other hand, the accuracy parameter is squared and dominates this equation, and therefore improving the accuracy is costly, as seen in the table. Improving accuracy by a factor of 2 requires 4 times as many samples (or in general, any improvement by a k factor requires k^2 as many samples). This effect is seen in Figure 2 for different confidence levels. While achieving an accuracy bound of up to 5% is fairly easy, achieving a 2% bound requires significantly more samples, and achieving a 1% assurance while providing an answer within the time constraints set by the use cases is unrealistic. The above formulation gives only a simple bound on the error, but in fact we know that for every fixed sample size the distribution of results will form a normal distribution around the actual ratio. Therefore, most results will actually lie very close to the actual result, far better than what the accuracy bound guarantees.

Note that if the data is mostly uniform in its compression ratio then we can stop the execution earlier (once it has been established that the observed samples have low variance). In real life however, the estimation was quick enough so that this additional test could be avoided.

Histograms and other estimations: The accuracy guarantees rely on the fact that the estimated value lies in a bounded range (in this case, between 0 and 1). Accurate estimation can therefore be assured also for other metrics that are similarly bounded. For instance, we can also estimate the fraction of the data that has a compression ratio within a certain range or the fraction of zero blocks in a volume. Indeed our tool outputs such histograms on compression ratio distribution within a volume (see Figure 5). On the other hand, if the estimation target is unbounded, then we cannot guarantee useful estimations. This is the case with the *running time of compression*, which has a high variance and is harder to estimate in a useful manner.

3.2.2 What to Sample and Compute

For the above accuracy analysis to be useful, we must be able to efficiently sample and compute the "contribution" of random bytes in the data set. To compute the contribution of a byte, or even to define it properly, the compression method must have some **bounded locality** properties; the compression of a given byte must be influenced by a limited number of other bytes. These other bytes should generally small in number and have offsets close to the byte in question. While straightforward for some methods, it is trickier for others, and may be altogether impossible for some compression techniques. To demonstrate this principle we give examples for the two types of compression that we handle:

Sampling in real-time compression: Real-time compression systems typically divide the data into chunks and compress each chunk separately. This is advantageous when data needs to be accessed randomly. Rather than decompress entire large files or large data segments, the decompression may be done locally on small chunks. Depending on the technique, these chunks may be fixed-size input chunks or alternatively may be defined by the actual data and/or its compression rate. Either way, the contribution of a specific byte is defined as the average contribution of the chunk that it belongs to. Accordingly, the sample phase will choose a random location and find the compression chunk surrounding it. The compression is then carried out only for this chunk, and the average compression for the chunk is the value added as the sample's contribution.

Sampling for zlib compression on objects: Another example of locality appears when running the DEFLATE algorithm on large data objects. Lempel-Ziv encodings are locally bounded due to practical considerations—back pointers are limited to 15 bits, allowing them to point to a window of only 32KB, and each repeating stream is limited to 256 bytes only in length (represented by 1 byte). Thus the compression ratio of a byte can be determined by examining a range of 32KB+256 bytes. Huffman coding, is also computed on a local window (determined by the space usage of the compression stream). Lower levels of DEFLATE will use a smaller window and thus easier to samlpe. In our estimation algorithm for the object interface we use the window size induced by the Lempel-Ziv encoding.

Once locality is understood for the particular algorithm, the question of how to define the *contribution* of a certain location remains. We first note that simply taking the average compression over a window of 32KB+256B does not fulfill our basic requirement that the overall ratio is an average of all the contributions. Indeed, this simple approach is misleading and can give gross miscalculations. The problem stems from the fact that a repeating string should be counted in full only on its very first appearance, as it has a very succinct representation on its second appearance. Therefore, given a chosen offset inside an object, we start running the compression process 32KB before this location, but discard the output. This pre-run serves as a warm-up to initialize the compression data structures to the state they would be at during compression of the object or volume. The contribution of the chosen offset is computed by the output size of compressing the next 256 bytes, divided by 256 (in other words, the average contribution to the output). Under this definition the average of contribution indeed sums up to the total compression ratio.

Note: Unlike the above mentioned compression examples, *deduplication* has no locality properties. In fact it can be viewed as a global version of compression, seeking repetitions across the entire data set. Indeed it was observed by Harnik et al. that deduplication ratios cannot be accurately estimated using sampling [10].

3.3 Implementation and Evaluation

We implemented the two versions of our estimator (the block device version and the objects version) in C++. The tool can spawn multiple threads, taking advantage of the inherent parallelism in processing random samples.

The block interface [13] estimates the compression ratio that would be achieved on a volume by using IBM's real-time compression product [22]. This is a host-based tool that can run on any storage attached to the host, and estimates two numbers: the percentage of *zero blocks* in the system, and the compression ratio on the non-zero blocks. We treat zero blocks as a special case, as

Data Type	Interface	Storage type	Size	Zero blocks	Exhaustive		Estimator	
					Run time	Ratio (%)	Run time	Ratio (%)
Development 1	real-time	Disk	279 GB	30%	4603 sec	17.28	177 sec	17.25
Development 2	real-time	Disk	279 GB	23%	5112 sec	28.20	204 sec	27.45
Development 3	real-time	Disk	279 GB	72%	4495 sec	87.09	207 sec	88.15
Development 4	real-time	Disk	137 GB	47%	2916 sec	13.45	150 sec	14.00
Hypervisor 1	real-time	Disk	3253 GB	49%	13800 sec	33.15	73 sec	32.99
Hypervisor 2	real-time	Disk	3253 GB	74%	19573 sec	18.24	140 sec	18.55
File repository	real-time	RAID5	390 GB	39%	2004 sec	50.16	38 sec	50.53
Object repository 1	object	RAID5	266 GB	–	6376 sec	40.93	21 sec	42.35
Object repository 2	object	RAID5	375 GB	–	10333 sec	51.60	44 sec	52.85

Table 2: Examples of estimator runs vs. an exhaustive calculation on various data sets.

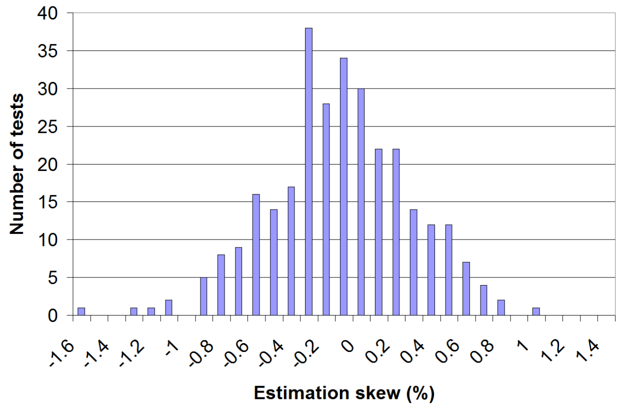


Figure 3: A histogram depicting the distance from the actual compression ratio for 300 runs of the macro-scale test.

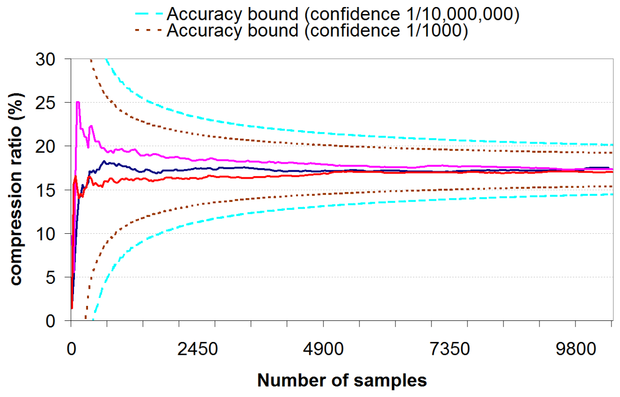


Figure 4: Three independent executions of the estimator converge on the true compression ratio as more samples are collected. The boundaries show the accuracy guarantees for each sample size.

this is generally free space that will be filled in the future, and we do not want to over-estimate the effectiveness of compression by including zero blocks that compress extremely well. Further, zero blocks generally do not take up actual space on modern storage systems due to thin provisioning. The performance of the estimator is highly dependent on the speed of the host machine (the tests were run on various x86 machines with access to the relevant storage), its connection to the storage, and the amount of parallelism supported by the storage (e.g., a single disk does not benefit from multi-threading, whereas RAID does). One last factor affecting performance is the fraction of zero blocks in the system. Zero blocks are not counted for the estimation, and therefore many more blocks need to be sampled in sparse volumes than in fuller volumes to achieve the required number of samples. For example, in a volume that is 90% empty, one needs to sample approximately $10x$ samples in order to find x non-zero blocks.

The **object interface** implementation takes as input a list of objects and their sizes. With such an interface, there is no issue with zero blocks.

We have tested our implementations on numerous data samples from active real-world machines to evaluate their speed and validate their accuracy. Table 2 presents a partial list of volumes on which our estimator was tested. They include several development machines running Linux, and two hypervisors with many virtual machine images that serve as a test bed for a large research project. In addition, we ran tests on an artificial file repository containing data from various applications, including, among others geo-seismic data, compressed and encrypted data, and databases to show that the estimator manages to handle highly heterogeneous data with varying compression rates (estimations on homogeneous data is far less challenging). All estimator tests ran within 3 minutes, while exhaustive tests ranged from half an hour to over five hours. Estimator performance greatly improves when running on a stronger machine (such as the hypervisors). The difference in running times between the two hypervisors is solely due to the higher fraction of zero blocks in the second machine. The best performance was achieved with RAID5 volumes, which benefit from multi-threading (all tests ran with 10 threads). This is extremely effective when running on enterprise storage systems (in such environments our estimator typically ran in well under one minute).

All tests were run with a minimum of 3,100 samples, enough to ensure an accuracy of 5.2% with confidence of 1 in 10 million (10^{-7}). The error in all tests was far lower than that, as can be seen by comparing the compression ratio achieved by the estimator with that of an exhaustive run. This is what we expect, as our analysis predicts that

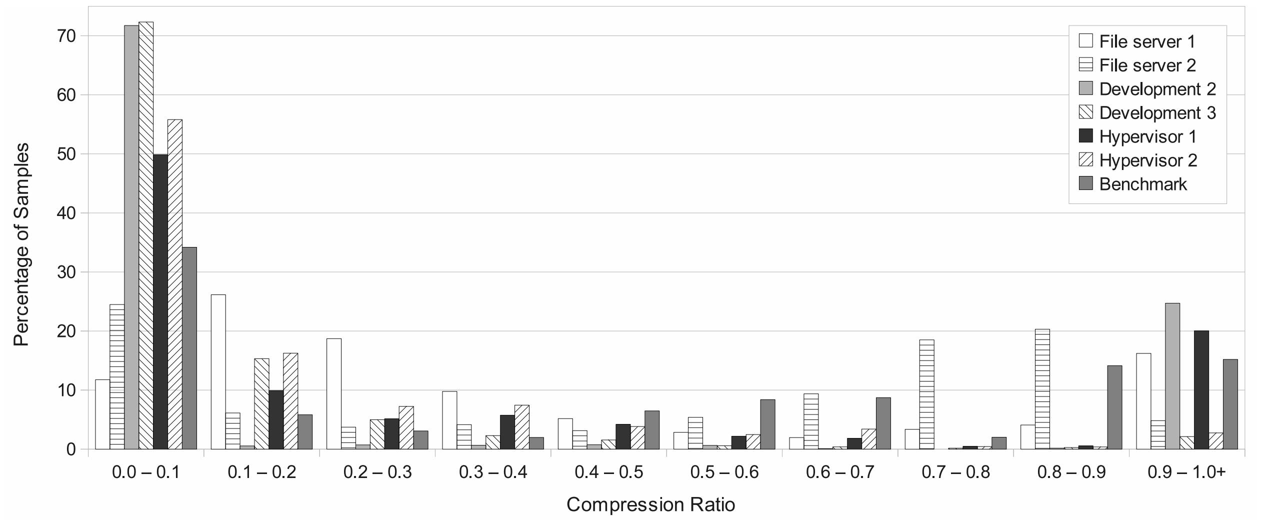


Figure 5: Compression ratio histogram for file server volumes. The histogram was sampled as described in Section 3.

the error is normally distributed around the true value, and most tests are likely to be very close to the center (true value). Figure 3 validates this analysis. This depicts 300 executions of the macro-scale test on the same data set and the result is a Gaussian distribution with nearly all of the mass is within the 1% boundaries. Figure 4 shows the convergence of three estimator executions as the number of samples increases. Theoretical accuracy bounds are shown as well, and the runs are well within these boundaries at all times.

Note that for the object interface, our sampling technique is affected by some implementation quirks in how zlib flushes the compressed data. Rather than modifying zlib, we give an external estimation that incurs a constant shift in the results of just over 1% (the estimator's results are slightly more pessimistic). This shift is noticeable in the object repository results in presented in Table 2.

4 Micro-Scale Compressibility

The decision to enable or disable compression at the level of an entire volume or file system can be too coarse-grained. Volumes containing data with varying levels of compressibility are common [16]. Consider documents with text and graphics; text is highly compressible, but graphics are typically pre-compressed. Many other examples are available: databases containing embedded BLOBs, virtual machine images, mail data files containing attachments, and Web server data containing both text and graphics.

We have identified many real-world use cases where compressing an entire volume would provide good capacity savings, although it contains a substantial amount of incompressible data. Figure 5 depicts examples of some real-world compression ratio histograms: two volumes from a file server (from the 42 volumes aggregately described in Table 3), as well as two hypervisor volumes and two development volumes (described in Table 2). The horizontal axis shows the compression ratio bins (lower is better), and the vertical axis denotes the percentage of samples with a given compression ratio. "Hypervisor 2" and "Development 3" are examples of volumes where most of the data is compressible, and so we would like to compress all data written to them. On the other hand, the other volumes shown contain much compressible data, but also at least 20% incompressible data. One would like to benefit from the capacity reduction of compression on such volumes, but also to manage the compression resources intelligently—compress data where compression provides a benefit, but skip the compression wherever it hardly achieves any space reduction.

The granularity at which we work is of single writes to the storage system. While these vary substantially in size, we focus on writes of size 8KB, 16KB and 32KB which are typical in primary storage systems. The most challenging case is of smaller writes (i.e., 8KB) where typically the overheads are higher compared to having few larger writes, and it is harder to evaluate compressibility without reading a majority of the write buffer.

The test data: The analysis of the micro-scale solution presented in this section is based on three data sets, whose characteristics are summarized in Table 3. The file server data set contains home or project directories owned by different users, stored in 42 different back-end volumes. The VM images data set is stored on eight back-end volumes. The file server and VM images data sets were sampled from active primary storage using the sampling techniques described in Section 3. The benchmark data set was artificially gathered to include many types of data with varying compression properties to test our algorithms. It includes images, documents (txt, csv, doc, xml, html, xls, pdf, ps), database files, VMWare images, geo-seismic data, call detail records, archival data,

Data Set	Total Size	Compressed Size	Compression Saving	Zero Chunks	Comments
File server	37TB	15.2TB	17.5TB	4.6TB	42 volumes, 144K samples
VM images	6.7TB	1.2TB	1.2TB	4.3TB	8 volumes, 22.9K samples
Benchmark	300GB	122.8GB	161.8GB	None	exhaustive scan

Table 3: Data sets characteristic summary.

compressed and encrypted files, and others.

4.1 Two Basic Approaches

We consider two approaches to testing compressibility:

Prefix estimation: A common method to estimate compression for files or large data chunks is to divide the data into segments and estimate by compressing one of the segments, typically the first one. Such an approach was suggested in [3] albeit at a different scale (consider compressing the first 1MB of each file). The decision to compress the data is based solely on the compressibility of the selected segment, and assumes that the sample is representative of the whole. Prefix estimation does not incur performance overheads when we decide to compress the data, as compressing the prefix is not wasted work. However, compressing the prefix is very wasteful when the sample is incompressible and we decide not to compress; not only will the compressed version of the sample not be used, but compression consumes the most resources when data is incompressible, as we have shown in in Figure 1. Another glaring shortcoming of this approach is in handling data that changes noticeably from its prefix (consider, for example data with a special header).

Heuristic based estimation: This approach refers to collecting simple heuristic parameters over the chunk that tend to characterize its compressibility and making the decision solely based on these heuristics. The heuristics must be extremely efficient, much more efficient than actually compressing the chunk. Due to this performance restriction, we collect our heuristics on random samples from the chunk rather than the whole chunk. Note that this efficiency improvement still circumvents the problem of the prefix method (a buffer that changes in the middle) since it samples the entire chunk.

An example of such a natural indicator for data compressibility is the *byte entropy* for the data [19]. Byte entropy is an accurate estimation of the benefits of an optimized Huffman encoding, and compression is generally more effective on buffers with lower entropy. Figure 6 depicts this correlation for the file server data set, where entropy and compression ratio were computed for 8KB data blocks. For this data set, an entropy of less than 5.5 typically predicts that the data will be compressible (compression ratio less than 0.8). However, the correlation is not ideal, as there is compressible data whose entropy is higher than 5.5. This is mainly due to the fact that entropy does not measure repetitions in the data, and therefore cannot capture all aspects of compression.

We have developed an estimation method that uses var-

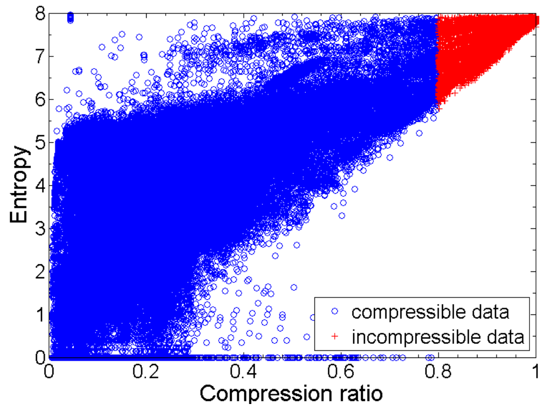


Figure 6: Byte Entropy as a function of the compression ratio of the file server data set.

ious heuristics to provide a more precise indication of compressibility, which we present in Section 4.2. In Section 4.5, we show how to combine this method with prefix estimation to exploit the best properties of each of the two approaches.

4.2 Our Heuristics

In this section we present our heuristic approach to determining compressibility. We considered a static threshold for the compression ratio above which data should not be compressed: 0.8. This threshold represents the trade-off between resource utilization and compression savings, and can be easily changed according to the amount of resources and expected load on the storage system. We use the following heuristics in our decision algorithm:

Data coreset size: We define the *coreset* to be the set of unique symbols that compose the majority (e.g., 90%) of the data. Logically, if the size of the coreset is small, we can expect benefits from the Huffman encoding as well as more repetitions and therefore the data is potentially a good candidate for compression. In contrast, a large coreset size indicates lower potential for obtaining benefits from compression.

Byte entropy: As mentioned above, byte entropy is a good indicator for compressibility of certain types.

Pairs distance from random distribution: The L_2 distance heuristic looks at pairs of symbols and the probability that the two symbols appear consecutively *in the coreset*. The distance is calculated as the (Euclidian) distance between the vector of the observed probability of a pairs of symbols appearing in the data, and the vector of the expected pair probabilities based on the (single) symbol histogram, assuming no correlation between subsequent pairs of symbols. The equation for calculating the L_2 is as follows:

$$\sum_{\forall a \neq b \in \text{coreset}} \left(\frac{freq(a) * freq(b)}{\text{size of sample}^2} - \frac{freq(a,b)}{\text{number of pairs}} \right)^2$$

This heuristic aims at distinguishing between randomly ordered data and data that contains repetitions. This allows us to distinguish between compressible and non-compressible data in cases where single byte heuristics such as entropy and coreset fails to do so.

Figures 7, 8 and 9 show each of the heuristics' potential thresholds for differentiating between compressible and incompressible data based on the data sets described in Table 3. We use this information to formulate the thresholds used in our algorithm.

In Figure 7 we see that for all three data sets, coreset size is directly related to compressibility. Based on this data we can set conservative values for when to compress (coreset size smaller than 50) and when not to compress (coreset size larger than 200). Figure 8 shows that an entropy lower than 6 is generally a good indication for compressible data. However, there is a portion of data with high entropy that compresses well. Figure 9 shows the distance from random distribution heuristic for data with an entropy higher than 5.5. This metric can be used together with entropy to further differentiate between compressible and incompressible data. Most of the incompressible data has a distance of less than 0.001 from the random distribution, while most of the compressible data has distance greater than 0.001 from it.

4.3 Implementation

The micro-scale approach works inside the storage I/O path and must incur very little overhead. To achieve the required low overhead we actually compute the heuristics on a sampled subset of the write buffer. In fact, this practice was also used when creating Figures 7, 8 and 9.

The size of the sample is based on a statistical sampling rule of thumb. It is common to perform sampling tests as long as the average number of elements in each bin (i.e., symbol) is at least five (for example, Chi-square tests). Instead of five, we choose eight as a more conservative value to obtain more accurate results. If the data contains the maximum number of symbols, which is 256, we obtain a sample size bound of 2048. Therefore, we sample at most 2KB of data per write buffer: 16 consecutive bytes from up to 128 randomly chosen locations. In practice, in most cases data write operations include far less than 256 unique symbols, allowing us to sample less data and improve the run time of the heuristics.

Our solution framework is composed of (1) random sampling a small subset of the write operation data, and (2) providing a recommendation on whether to compress

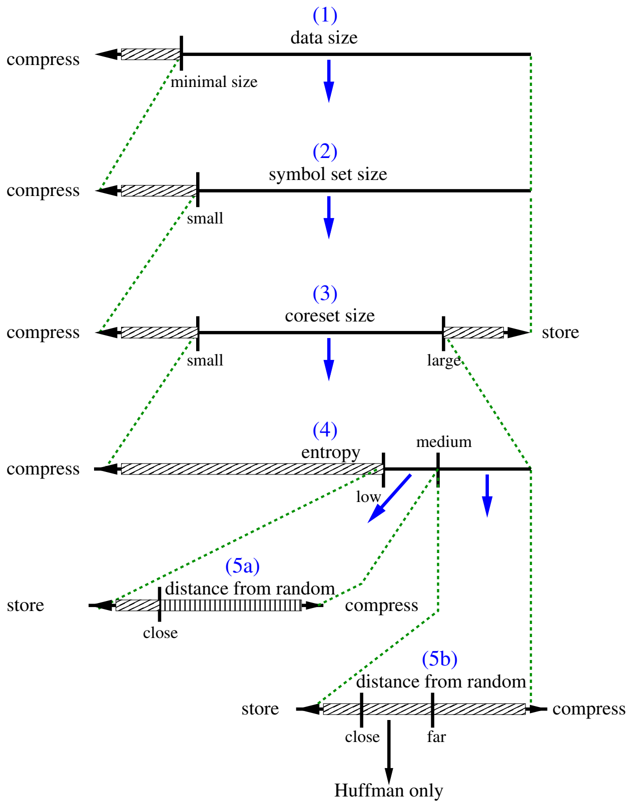
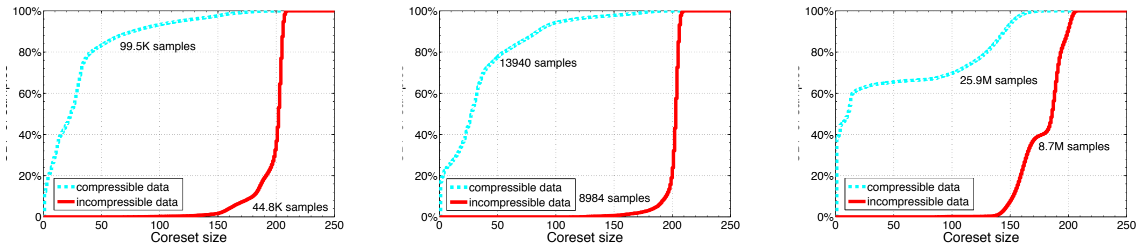


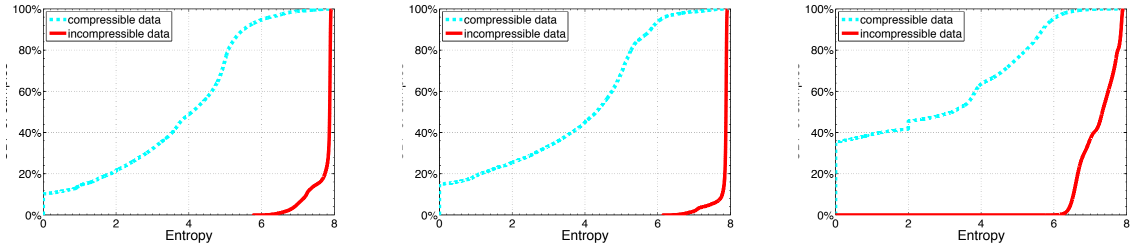
Figure 10: The algorithm for estimating data compressibility using heuristics.

or not compress the data based on the heuristics in Section 4.2. The recommendation algorithm is outlined in Figure 10. The algorithm flow is from top to bottom, and at each step one parameter or heuristic is examined. A recommendation is made if the heuristic satisfies the given thresholds. The horizontal arrows represent recommendations and the downward arrows represent moving to the next heuristic computation. The vertical bars represent the thresholds for making the recommendation at each step. The algorithm is designed for speed, the heuristics are ordered according to their computation time, from light to heavy calculation and a recommendation is made as early as possible, reducing the computation overhead. The algorithm is outlined top to bottom follows: **(1)** Small amounts of data (smaller than 1KB) should always be compressed, as calculating the heuristics will generally take longer than compressing. **(2)** If the total number of symbols in the data is very small, e.g., 50, then compress. **(3)** If the coreset size is very small (e.g., smaller than 50), compress, and if it is very large (e.g., larger than 200), store (do not compress). **(4)** If the entropy is reasonably low (e.g., smaller than 5.5), compress. **(5a)** Data with medium entropy (e.e., 5.5–6.5) and a small distance from random distribution (e.g., 0.001) should be stored; for higher distances, compress. **(5b)** Data with higher entropy (e.g., greater than 6.5) and a small distance from random distribution



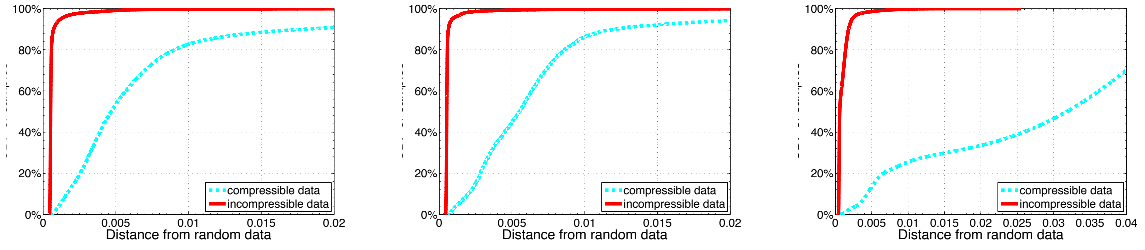
(a) File server data set. (b) VM images data set. (c) Benchmark data set.

Figure 7: Coreset size CDFs for compressible and incompressible data.



(a) File server data set. (b) VM images. (c) Benchmark data set.

Figure 8: Entropy CDFs for compressible and incompressible data.



(a) File server data set. (b) VM images data set. (c) Benchmark data set.

Figure 9: Distance from random distribution CDFs for compressible and incompressible data with entropy higher than 5.5.

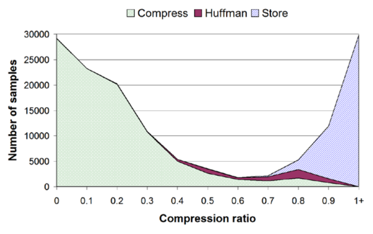
(e.g., 0.001) should not be compressed; data relatively far enough from random distribution (e.g., 0.02) should be compressed. Data in between the distance thresholds should be compressed using Huffman coding; it provides a balance when the heuristics do not provide a clear decision. We can see that the majority of the data for which the heuristics recommend to use Huffman coding falls into the incompressible bins, or very close to the incompressible bins.

4.4 Evaluation

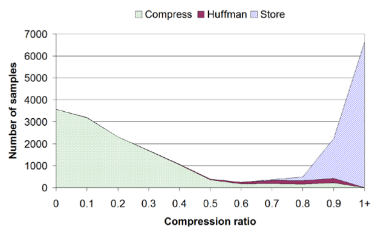
We evaluated the run time and compression performance of the prefix estimation and the heuristics method on a dual processor Intel Xeon L5410 (2.33Ghz) Linux server. Note that our implementation is single threaded and did not exploit the system parallelism. The methods' output is either to compress the data (we used zlib level 1 for this) or store it without compression (using zlib level 0 to copy it). The heuristic method can also recommend using Huffman coding only (using the appropriate zlib flag). Our measurements do not include the time for performing the disk read and write operations, as our focus is on the CPU resources for compression and the resulting capacity impact. The evaluation focuses on two categories: (1) accuracy of compressibility estimation, and (2) time and capacity impacts.

Figure 11 shows the heuristic method's recommendations for each compression ratio bins for the three test data sets listed in Table 3. We see that the overwhelming majority of the incompressible data with compression ratio higher than 0.9 are identified as such. Compressible data (with compression rate under 0.8) is identified almost always as compressible. For compression ratios between 0.8 and 0.9 the recommendations are mixed. Recall that the intention was to categorize such data as incompressible but data in this range turns out to be difficult to identify accurately (especially for 8KB blocks).

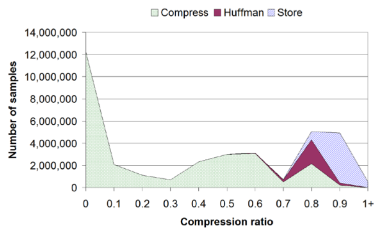
Next we examine the run time of the different methods as tested on 8KB blocks. Figure 12 shows the run time of the prefix method on a 1KB prefix, of the heuristic method and this is contrasted with compression time of the entire 8KB data block. The heuristic approach is very fast both for data with very good compression and for highly non-compressible data. For mid-way data, with



(a) File server data set.



(b) VM images data set.



(c) Benchmark data set.

Figure 11: The heuristic approach recommendation by compression ratio for 8KB data blocks.

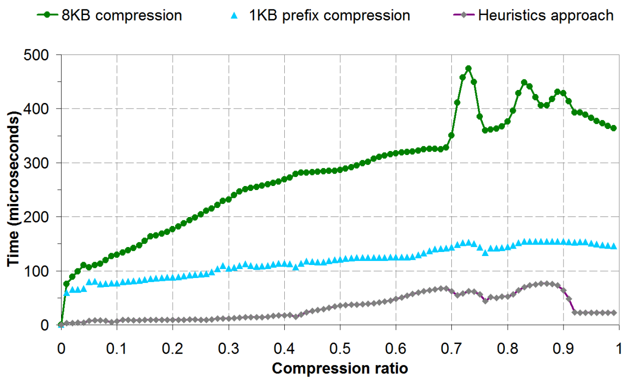


Figure 12: Comparison of average compression time of compressing a 1KB prefix, compressing the whole 8KB data, and the heuristics approach for data of various compression ratio.

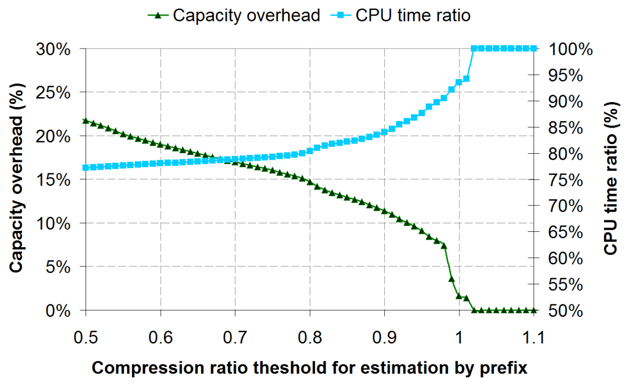


Figure 13: Time and capacity impact of compressing 8KB data blocks from the benchmark data set using prefix estimation with different compression ratio compressibility thresholds.

compression ratios of 0.4 through 0.9, it needs to review all (or almost all) the heuristics to reach a recommendation and thus requires a longer running time. For all compression ratios, the heuristic approach is much faster than running the prefix estimation. However, for compressible data (whenever the correct decision is made) the prefix estimation essentially comes for free, as we can just continue to compress the data from the 1KB point. For incompressible data, on the other hand, the prefix compression estimation is plainly an overhead. We can therefore conclude that that the prefix method is slightly better for compressible data, while the heuristic method is faster by $2X$ to $6X$ for non-compressible data.

Figure 13 focuses on the prefix method and examines the impact of selecting different compression ratio thresholds for differentiating between compressible and incompressible 8KB data blocks. For each such choice we depict the impacts on capacity and CPU time as measured on the benchmark data set. The horizontal axis is the threshold used to identify incompressible data; for example 0.9 means that if the compression ratio of the prefix was higher than 0.9 the data will not be compressed. Selecting a threshold of 0.99 inflates the capacity by 3.6% yet utilizes only 92% of the CPU time required had all the data been compressed.

In Table 4 we compare the CPU and capacity impacts of the the heuristic method versus the prefix estimation when compressing block sizes of 8KB, 16KB and 32KB from our benchmark data set. For the prefix estimation we present two different thresholds: the first matches the capacity impact of the heuristic method and then the CPU savings can be measured on equal terms. The second matches the CPU savings of the heuristic and then the capacity can be examined. Note that there was no single threshold for the prefix estimation that could match or surpass both the CPU time and capacity overhead.

The heuristic approach manages to consistently limit the impact on capacity (about 2%) for all block size and shows a reduction of CPU time as the block sizes increase. On the other hand, the prefix estimation needs to sacrifice either capacity or CPU time to match the heuristics approach. To match the CPU time, the prefix estimation loses between 10.4% (for 8KB) to 4.4% (for 32KB) of capacity compared to only about 2%. To match the 2% capacity overhead, the prefix estimation speedup is between 92% (for 8KB) and 74% (for 32KB) compared to a speedup of between 85% (for 8KB) and 65% (for 32KB) when using the heuristic approach.

It is important to note that the benchmark data set contain about 30% incompressible data and 70% compressible data (the compression ratio histogram are available in Figure 5). For data sets with higher portion of incompressible data, the heuristics approach will provide greater benefit and will in fact increase the performance gap between the prefix estimation and the heuristics approach. On the other hand, as the portion of compressible data grows, the prefix method will become more suitable.

Block Size	Method	Capacity Overhead	CPU Time
8K	Heuristics	2.0%	85%
	1K prefix w/ 0.99	3.6%	92%
	1K prefix w/ 0.92	10.4%	85%
16K	Heuristics	2.3%	74%
	2K prefix w/ 0.95	1.8%	86%
	2K prefix w/ 0.87	7.0%	74%
32K	Heuristics	2.3%	65%
	4K prefix w/ 0.88	2.2%	74%
	4K prefix w/ 0.82	4.4%	65%

Table 4: Comparing CPU time ratio and capacity impact of prefix estimation versus the heuristics approach on the benchmark data set.

4.5 Putting it All Together

While the heuristic approach has noticeable advantages when there is a significant amount of incompressible data on the storage volume, this is not the case when nearly all of the data is compressible. In such cases the heuristic only adds an overhead to the run time without any gain.

This calls for an adaptive on-demand deployment of the estimation techniques. We propose to employ both methods, the prefix estimation and the heuristics method, within a single solution. Basically, employ the prefix method when all or most of the data is compressible, but switch to the heuristics approach whenever enough non-compressible data is encountered. This mixed approach introduces only minimal overheads when handling mostly compressible data, but will provide great CPU relief once incompressible data is encountered.

Moreover, consider workloads for which there is a clear distinction between times when incompressible data is written to periods of compressible data writes. For example, periods during which encrypted or zipped data are written. In such scenarios, switching back and forth between prefix estimation and heuristics will deliver optimal performance.

An additional opportunity for adaptiveness is in the heuristics thresholds. We strived to collect as many real world data types in our benchmark data set and suggested thresholds that perform well on the various data sets. However, we suggest that the thresholds be adaptive to the data encountered and fine tuned during execution to optimize usage of both capacity and CPU time.

5 Related Work

NetApp provides the Space Savings Estimation Tool (SSET) to estimate the benefits of deduplication and compression on a portion of a file system. Whereas our macro-scale solution provides accuracy guarantees necessary for making decisions with confidence, NetApp claims that "in general, the actual results are within $\pm 5\%$ of the space savings that the tool predicts" [18].

Estimating overall compression savings when compressing each file has been explored in [3]. The authors experimented with compressing a set of randomly-sampled files, but saw that this resulted in a high variance between sample sets. The method they chose is to compress the first megabyte of the largest files in the file system, which comprise some percentage of the total capacity. Their method requires a full file system scan and sort, and results show that between 0.02% and 1.93% of the file system needs to be compressed for stable results. They do not provide any statistical guarantees.

There is a procedure in Microsoft SQL Server called `sp_estimate_data_compression_savings` that estimates the compression savings for a given table [17]. Little is stated about its implementation, other than that a sample is compressed. No accuracy guarantees are given, and some users report up to a 20% error [1].

Harnik et al. discuss how to piggyback compression estimation to a process of estimating deduplication ratios [10]. However, the authors do not discuss a stand-alone algorithm for compression estimation.

Several works attempt to determine the most suitable compression algorithm for a given chunk of data. One such work concluded that the standard deviation of the bytes is a good predictor for Huffman encoding effectiveness, while the standard deviations of the difference of consecutive bytes and XORed value of consecutive bytes can be used to predict the effectiveness of 12-bit LZW [4]. However, statistics were gathered on entire files, and only correlations were shown, with no discussion on performance or using the metrics together to determine compression effectiveness. Another uses genetic programming to attempt to predict exact compression ratios using the byte frequency distribution and features extracted from the data stream [14]. This method is not suitable for our use case, as it spends a significant amount of time finding an exact answer, whereas we require a fast indication of compression effectiveness.

6 Concluding Remarks

We have shown how to effectively utilize storage system resources for real-time compression by efficiently and accurately filtering out incompressible data. The techniques we demonstrated allow the storage system to invest its valuable resources only in data that compresses well. Rather than compressing incompressible data, the resources are now free to compress other data as well as to serve I/O requests, thus improving both cost savings and performance.

Acknowledgments: We would like to thank our colleagues at the IBM Real-time Compression team and the Haifa Research lab for their involvement and support of this work. Specifically we thank Jonathan Amit, Lilia Demidov, Olga Dodin, Nir Friedman, George Goldberg, Nir Halowani, Chaim Koifman, Ido Levy, Sergey Marenkov, Dalit Naor, Orit Nissan, Tzahi Shahak and Assaf Yaari.

References

[1] E. Bertrand. Playing with page compression - for real. `http://sqlblog.com/blogs/aaron_bertrand/archive/2009/12/29/playing-with-page-compression-for-real.aspx`.

[2] Michael Burrows, Charles Jerian, Butler W. Lampson, and Timothy Mann. On-line data compression in a log-structured file system. In Barry Flahive and Richard L. Wexelblat, editors, *ASPLOS*, pages 2–9. ACM Press, 1992.

[3] C. Constantinescu and M. Lu. Quick Estimation of Data Compression and De-duplication for Large Storage Systems. In *Proceedings of the 2011 First International Conference on Data Compression, Communications and Processing*, pages 98–102. IEEE, 2011.

[4] W. Culhane. Statistical Measures as Predictors of Compression Savings. The Ohio State University, Department of Computer Science and Engineering, Honors Thesis, May 2008.

[5] P. Deutsch and J. L. Gailly. Zlib Compressed Data Format Specification version 3.3. Technical Report RFC 1950, Network Working Group, May 1996.

[6] EMC. EMC Data Compression: A Detailed Review. Technical Report h8045.1, EMC, September 2010.

[7] J. L. Gailly and M. Adler. The Zlib home page. `www.gzip.org/zlib`, 1998.

[8] J. Gantz and D. Reinsel. 2009 Digital Universe Study: As the Economy Contracts, the Digital Universe Expands. Technical Report White paper, International Data Corporation, May 2009.

[9] J. Gantz and D. Reinsel. 2011 Digital Universe Study: Extracting Value from Chaos. Technical Report White paper, International Data Corporation, June 2011.

[10] D. Harnik, O. Margalit, D. Naor, D. Sotnikov, and G. Vernik. Estimation of Deduplication Ratios in Large Data Sets. In *Proceedings of the 18th International IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2012.

[11] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 301(58):13–30, 1963.

[12] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[13] IBM. IBM Comprestimator Utility V1.2.0.2 - United States. `http://www-01.ibm.com/support/docview.wss?uid=ssg1S4001012`.

[14] A. Kattan and R. Poli. Genetic-Programming Based Prediction of Data Compression Saving. *Lecture Notes in Computer Science*, 5975/2010:182–193, 2010.

[15] D. Kay. Oracle Solaris ZFS Storage Management. Technical Report 507914, Oracle Corporation, November 2011.

[16] M. Lu, D. Chambliss, J. Glider, and C. Constantinescu. Insights for Data Reduction in Primary Storage: a Practical Analysis. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR 2012)*. ACM, 2012.

[17] Microsoft. sp_estimate_data_compression_savings (Transact-SQL). `http://msdn.microsoft.com/en-us/library/cc280574.aspx`.

[18] S. Moulton and C. Alvarez. NetApp Data Compression and Deduplication Deployment and Implementation Guide: Data ONTAP Operating in Cluster-Mode. Technical Report TR-3966, NetApp, June 2012.

[19] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423, 1948.

[20] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST 2012)*. USENIX Association, 2012.

[21] Nimble Storage. Nimble Storage: Engineered for Efficiency. Technical Report WP-EFE-0812, Nimble Storage, 2012.

[22] J. Tate, B. Tuv-El, J. Quintal, E. Traitel, and B. Whyte. Real-time Compression in SAN Volume Controller and Storwize V7000. Technical Report REDP-4859-00, IBM, August 2012.

[23] R. Tretau, M. Miletic, S. Pemberton, T. Provost, and T. Setiawan. Introduction to IBM Real-time Compression Appliances. Technical Report SG24-7953-01, IBM, January 2012.

[24] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(2):337–343, May 1977.

[25] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978.