Scaling Down Off-The-Shelf Data Compression: Backwards-Compatible Fine-Grain Mixing

Michael Gray, Peter Peterson, Peter Reiher
Computer Science Department
University of California, Los Angeles
Los Angeles, USA
{megray, pahp, reiher}@cs.ucla.edu

***Abstract*—Pu and Singaravelu presented Fine-Grain Mixing, an adaptive compression system which aimed to maximize CPU and network utilization simultaneously by splitting a network stream into a mixture of compressed and uncompressed blocks. Blocks were compressed opportunistically in a send buffer; they compressed as many blocks as they could without becoming a bottleneck. They successfully utilized all available CPU and network bandwidth even on high speed connections. In addition, they noted much greater throughput than previous adaptive compression systems. Here, we take a different view of FG-Mixing than was taken by Pu and Singaravelu and give another explanation for its high performance: that fine-grain mixing of compressed and uncompressed blocks enables off-the-shelf compressors to scale down their degree of compression linearly with decreasing CPU usage. Exploring the scaling behavior in-depth allows us to make a variety of improvements to fine-grain mixed compression: better compression ratios for a given level of CPU consumption, a wider range of data reduction and CPU cost options, and parallelized compression to take advantage of multi-core CPUs. We make full compatibility with the ubiquitous deflate decompressor (as used in many network protocols directly, or as the back-end of the gzip and Zip formats) a primary goal, rather than using a special, incompatible protocol as in the original implementation of FG-Mixing. Moreover, we show that the benefits of fine-grain mixing are retained by our compatible version.**

I. Introduction

For network transmission, the benefit of data compression is often outweighed by its cost. While compression can dramatically increase throughput under ideal circumstances, higher than expected bandwidth or lower than expected CPU availability can make compression a bottleneck rather than an optimization. As a result, it is often confined to systems with predictable and stable workloads, or like in OpenSSH[1], is off-by-default and must be enabled manually.

Researchers have attempted to improve this situation with "adaptive compression" techniques designed to make ideal compression decisions dynamically in the face of varying conditions. A number of adaptive compression schemes[2], [3], [4], [5] have attempted to accomplish this by switching compression algorithms, turning compression on or off, or tuning compressor "strength" settings in an attempt to compress "just enough" in the face of changing network, CPU, or data characteristics.

Unfortunately, the strength parameters of typical compressors provide far from linear scaling. In reality, increasing from minimum to maximum strength with common compressors such as deflate[6] offers only a small increase in data reduction while incurring a large CPU penalty[7]. Selecting among compression algorithms can provide a broader range of initial performance values, but is more complicated and still fails to provide a real spectrum of compression ratio and CPU costs between those algorithms. Additionally, there is usually a significant difference (in terms of compression ratio and the CPU time required) between "no compression" and the "cheapest" option, regardless of the algorithm chosen.

When the choice whether to compress or not is obvious, adaptive compression schemes can work well. Yet because of the significant difference between these two options, this "simple switching" strategy can result in the underutilization of either the CPU or network[8], even when the "right" choice is made. If no compression is the best choice, the CPU will be left idle. If compressing is the best choice, "overcompression" may reduce data unnecessarily, wasting CPU time and underutilizing bandwidth.

Pu and Singaravelu identified this problem and introduced the concept of Fine-Grain Mixing (FG-Mixing) for adaptive compression in dynamically varying networks[8]. Rather than trying to pick the best single strategy (potentially underusing some resource), FG-Mixing implemented an innovative scheme to simultaneously maximize the use of both network and CPU[8]. Their system divided the outgoing buffer for a network stream into blocks and compressed as many of those blocks as possible without delaying transmission. Any remaining blocks were sent uncompressed. As available CPU increased (more cycles for compression), or network bandwidth decreased (relatively more time for compression), a greater proportion of blocks in the outgoing buffer could be compressed, improving effective throughput. Similarly, when available CPU decreased (less cycles for compression), or network bandwidth increased (relatively less time for compression), a greater proportion of the blocks

would simply be transmitted uncompressed.

Pu and Singaravelu's solution was able to maximize both CPU and bandwidth usage, successfully scaling from the network constrained case where compressing all blocks improved throughput, up to 400 megabits per second where the majority of blocks were sent uncompressed. Their implementation significantly outperformed a model simple switching approach, both in terms of resource utilization and throughput. They also demonstrated that the concept of FG-Mixing generalizes to many algorithms by performing mixing experiments with a selection of compressors, including the GNU Gzip implementation of deflate and bzip2.

We explore a different view of FG-Mixing and give another explanation for its flexibility and performance: that — unlike legacy strength options — fine-grain mixing of compressed and uncompressed blocks gives inflexible legacy compressors a continuous and responsive "knob" for scaling compression linearly with respect to CPU usage. Additionally, this reliable control allows bandwidth use to be maximized opportunistically — increasing when it is profitable, decreasing when it is not — rather than relying on predictive models. This alternative view suggests a variety of significant efficiency improvements to the original work that result in better compression ratios for a given level of CPU consumption, a wider range of data reduction and CPU cost options, and parallelization to take advantage of multi-core CPUs.

Additionally, Pu & Singaravelu's FG-Mixing implementation used a custom protocol for transmitting the mixture of compressed and uncompressed data, requiring non-standard decompressors. Instead, we show that fine-grain mixed compression can be implemented without performance degradation while remaining fully backwards-compatible with the ubiquitous deflate specification. This means that off-the-shelf systems employing deflate-based compression could benefit from fine-grain mixing without significant re-engineering or broken compatibility. This virtually eliminates the technical obstacles to FG-Mixing's adoption in existing, heterogeneous, and widely distributed systems.

II. Fine-Grain Mixing for Scalable Compression

A. Backwards Compatible FG-Mixing with Deflate

A primary objective of this project is to generate fine-grain mixed streams that are backwards-compatible with off-the-shelf deflate decompressors. A backwards-compatible stream allows systems to use FG-Mixing at the transmitting side, while utilizing any compliant deflate decompressor at the receiving end (if data is flowing in two directions, one or both sides may use FG-Mixing independently of whether the other transmitter supports it). In contrast, all of the other adaptive compression systems mentioned previously used proprietary formats or protocols in their implementation, which limited their usability outside of special-purpose applications.

Deflate[6] is a popular, byte-stream oriented compressed data format employing a combination of the LZ77 dictionary-based compression algorithm and Huffman coding. It is hard to overstate how widely deployed deflate compression is in the modern computing environment. The stream itself is often used directly, but is more commonly wrapped in a format such as zlib[9] or gzip[10], which provide additional metadata and integrity checking, or is used in compressed archive formats like Zip[11]. While the ubiquitous Zip format supports many compression methods[11], deflate streams are so common that some Zip expanders only support deflate. In addition to popular compression formats, deflate is officially part of many applications and protocols, such as PPP, SSH, HTTP, RSYNC, and PNG[12], [13], [14], [11], [10]. Deflate is so widely used that it is sometimes even accelerated in hardware[15].

The deflate specification describes a data format rather than a particular encoding algorithm in order to allow for reduced-resource or alternative compressors. Nevertheless, deflate assumes the LZ77 algorithm and Huffman coding will be employed in some form. Huffman coding is a form of entropy coding which maps each symbol to a variable-length code based its real or expected frequency in the output[16]. LZ77 is a dictionary-based algorithm that maintains a sliding window over the historically encoded data[17].

A stream of data compressed with deflate is composed of three types of blocks: compressed blocks encoded with Huffman trees built in to the specification, compressed blocks encoded by Huffman trees included within the stream, and uncompressed blocks. Compressed blocks can be of unlimited length, but very large blocks are not always beneficial and may be broken into smaller output blocks. Uncompressed blocks can each contain up to 64KB, with their length being specified in a block header. Uncompressed blocks can occur naturally, for example, when LZ77 and Huffman-encoding a block would lead to expansion of the input. These three types of deflate stream blocks will be referred to as "deflate-blocks" henceforth to avoid confusion with higher-level "blocks" (such as input data segments).

As described above, deflate is very widely deployed, and deflate streams consist — by specification — of mixtures of compressed and uncompressed blocks in any proportion. Our implementation will take advantage of these native, uncompressed deflate-blocks to achieve backwards-compatible fine-grain mixed compression.

III. Implementation

In order to pursue our investigation of the scaling properties of a deflate-compatible, scalable, parallel implementation of FG-Mixing, we created Mpigz. Mpigz is a heavily modified version of pigz[18], a parallel implementation of gzip employing the zlib compression library. We used pigz 2.1.6 (linked with zlib 1.2.5), making variety of major

and minor changes. Significant changes and implementation details are described in the following sections.

A. Compressor Initialization

Pu and Singaravelu's FG-Mixing implementation treated input blocks independently. In other words, each new block of input was compressed in isolation — requiring the compression history used for identifying redundancy to be refilled on the fly. However, pigz supports sharing history from block to block (we call this "shared history" as opposed to "independent" mode). This mode is slightly more expensive in terms of CPU, but can improve compression significantly because the history for each block is "warmed" by the compression of previous (and ostensibly similar) input. While pigz compresses in "shared history" mode by default, it provides a switch to treat blocks independently, which we used to compare these two options.

B. Fine- and Finer-Grain Mixing

A fine-grain mixing option was added to specify the mixture percentage of two compression settings. In order to ensure repeatability of the tests, the mixing procedure was made deterministic. Input blocks are split into groups of 100, the first pp (where pp is the mixing percentage) are set to use "weak" compression, and the remaining $100 - pp$ are set to use "strong" compression. They are then shuffled pseudo-randomly with a fixed seed. A real-world implementation might base decisions on a random boolean, true pp percent of the time.

Additionally, while FG-Mixing only mixed uncompressed blocks with blocks compressed at a single strength level, we wanted to explore the scaling properties of mixing both "weakly" and "strongly" compressed blocks. Thus, options were added to explicitly set the compression method for both "weak" and "strong" blocks. The standard zlib strengths 0 (no compression) through 9 (maximum compression) are supported, in addition to three special zlib reduced-strength strategies: Filtered, RLE (run length encoding), and Huffman-Only. These special modes are intended to provide less compression, but greater throughput, than a full deflate implementation. Because they are intended for specific types of data, they are not necessarily as general purpose as the full implementation. Nevertheless, they generate compatible streams, so we felt they were worth investigating.[1]

C. Efficient Direct Generation

Fine-grain mixing implementations require a fast method for generating uncompressed blocks, because they allow FG-Mixing to "catch up" when there is not enough CPU time to compress all the data. Zlib generates uncompressed blocks when the strength is set to 0, but we found it surprisingly slow. Because we were performing tests generating the gzip format, which uses CRC-32 for integrity checking, we assumed that checksum generation would account for most of the CPU use of strength 0 compression. Instead, only 13% of execution time was spent generating the CRC-32, while the remaining 86% was spent in zlib itself.

Two components account for the time spent in zlib. 34% of total execution time was spent copying the data in an inefficient bytewise fashion, while 52% was spent in a routine updating hash-tables for LZ77. However, LZ77 is obviously not performed when emitting uncompressed blocks! A comment in the zlib source code indicates that the LZ77 hash tables are being updated in case the client code ever re-enables compression, referring to persistent use of uncompressed mode as a "pathological case."

For our purposes, strength 0 is a "common case," and the compressor will always be reinitialized when starting a new (and possibly more strongly compressed) block, so there is no reason to waste time updating structures for the unused LZ77 algorithm. Instead, we wrote our own direct generation routine (DG), which uses only 7% of execution time (compared to 86% used by zlib), which is almost entirely spent calling `memcpy`. This allowed 91% of execution time to go towards generating the CRC-32 (versus 13% with zlib), radically improving the throughput of uncompressed block generation.

D. Single-threaded Block Mode

Pigz uses zlib to perform deflate compression in a multi-threaded fashion. Normally, when only one thread is requested, pigz uses a single thread to consume and compress data in a bytewise (non-blocked) fashion with a sliding history window. Blocking mode is enabled when two or more compressor threads are requested. Additionally, when blocking mode is enabled, a reader thread splits input into blocks (of default size 128KB) to be consumed by the compressor threads, and a writer thread combines any necessary checksum values from the compressed blocks and orders the combined output.

Our implementation depends on the block-based behavior of pigz's multi-threaded mode for fine-grain mixing, but we wanted to compare the performance of Mpigz with single-threaded legacy utilities. Accordingly, we modified pigz so that by default, when one thread is requested, it runs in the block-based mode (rather than the bytewise mode) but with one compression thread. Even though we use a single compression thread, the pigz architecture still requires separate reader and writer threads in addition to the compressor thread. In order to accurately characterize the total computational cost in our evaluation, we typically report user times (the sum of all CPU times consumed by each CPU core) rather than "real" time, to allow for direct comparison with single-threaded implementations.

[1] While these reduced-strength modes were intended to be higher-speed, they were actually quite slow until zlib 1.2.4 (released in March of 2010). Our Linux distribution (Ubuntu 11.04) includes zlib 1.2.3.4, so we manually installed the new versions. Interestingly, these strategies were tested with adaptive compression in 1999[4], but were limited by the older code.

IV. Evaluation

A. Hardware and Methodology

All scaling evaluation was performed on a dedicated system with an Intel E8200 Core 2 Duo at 2.66GHz and 3GB of RAM running Ubuntu Server 11.04 for AMD64. CPU frequency scaling and non-essential background tasks were disabled. Unless otherwise specified, all tests used 128KB input blocks compressed using "shared history" mode. All output was generated in the gzip format, a realistic choice, but with a slightly-higher CPU cost than alternatives (such as a bare deflate stream) because of the CRC-32 used for integrity checking. Correctness was verified by decompressing a variety of Mpigz output with multiple unmodified legacy decompressors.

For performance testing, input was read from a ramdisk and output was discarded to remove the influence of disk I/O speed on the results. Each point in a chart represents the mean of 5 runs of a given experiment. Following a cache-warming dummy run (which was excluded from the results), the individual component tasks of an experiment were executed in a random sequence to reduce ordering bias. All reported times are "user times" as described in Section III-D.

The term "reduction" refers to the difference between the size of the uncompressed and compressed data, expressed as a percentage of the input size. For example, a reduction of 60% indicates that compressed file was 40% of the size of the original. Error bars, which are present unless noted, represent plus or minus one standard deviation, but are often too small to be visible.

B. Standard Workload

While no single workload can be representative of all possible inputs, we felt that a single, realistic workload could be used to explore the basic performance of our technique. We chose to use the uncompressed filesystem image from the Ubuntu 11.04 Desktop for AMD64 Linux distribution. This widely-available image includes a variety of data types; it is used as the "live" image when booted from a CD and as the install image to be copied to a hard drive for permanent installation. It is thus representative of a fully-functional desktop operating system.

Such an workload might represent a common use case for adaptive compression: a user backing up their portable computer to their home over varying network connectivity. This user might backup rarely over a slow connection while traveling, over a wired LAN when home, or over a wireless connection of fluctuating bandwidth. Since we believe this to be a reasonable workload providing a large and realistic mixture of data types, we use this image as the standard input throughout this paper. See Section IV-C2 for further justification.

C. Experiments

1) Comparing Mpigz and GNU Gzip: We wanted to ensure that Mpigz produced results comparable to a mature deflate implementation. Thus, we compared the non-mixed scaling behavior of Mpigz with the GNU implementation of Gzip. We chose GNU Gzip because many other implementations of Gzip (such as those included with FreeBSD and derivatives) are actually command-line wrappers for zlib[19], which itself Mpigz employs.

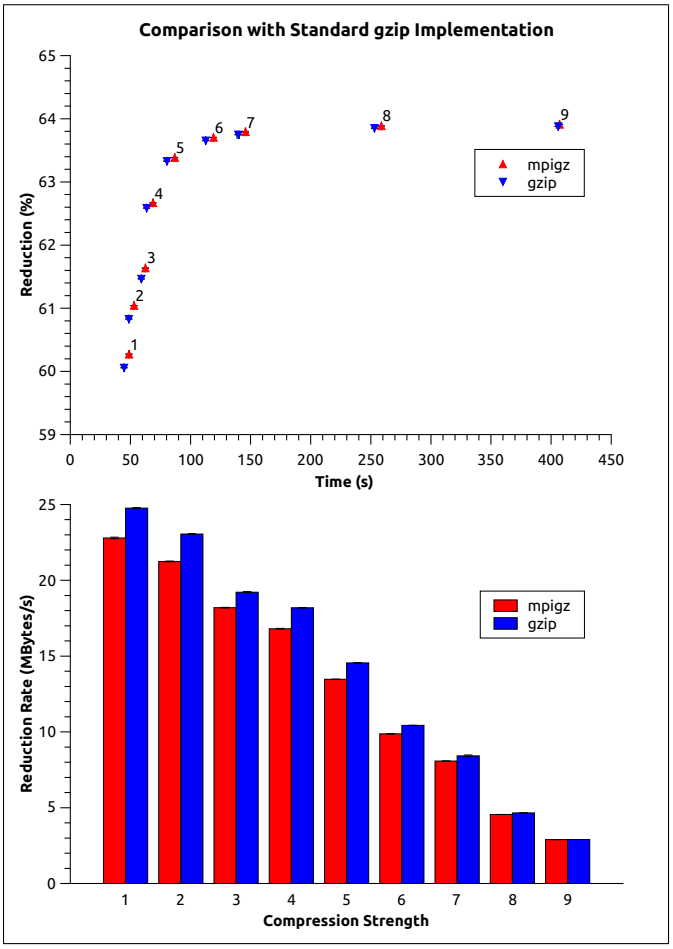

Figure 1. Comparison of Mpigz versus unmodified GNU Gzip.

The top chart in Figure 1 plots the size reduction provided versus the CPU time consumed by varying compression strengths for both GNU Gzip and Mpigz. The scaling behavior is extremely similar. Mpigz compresses slightly more at the expense of consuming slightly more CPU time at the lower strengths, but these differences diminish towards the higher-strength compression options.

The bottom chart in Figure 1 shows the "reduction rate" — how many megabytes in size the data is reduced per second of consumed CPU time — as we increase the compression strength for both GNU Gzip and Mpigz. We can think of reduction rate as a general measure of efficiency for the

purposes of adaptive compression, where the compression strategy is determined based on the amount of reduction that is necessary in a given amount of time in order to provide a benefit. The reduction rate is equivalent to the slope of a line starting at the origin and passing through the points in the upper portion of Figure 1 (if the Y-axis were expressed in megabytes rather than percentages). Again, we see that while at the lower strength settings Mpigz's reduction rate is slightly lower (approximately 23 megabytes per second for Mpigz versus 25 megabytes per second for Gzip), the two strategies grow to become essentially equivalent by the highest setting. Since the performance of Mpigz and GNU Gzip is overall very similar in these tests, we assume that the basic behavior of Mpigz is comparable to mature off-the-shelf compressors.

2) Deflate Scaling by Strength and Data Type: Mpigz and GNU Gzip have comparable non-mixing performance for our standard workload. However, the reduction achieved by an algorithm can vary significantly based on input characteristics. To verify that our standard input compresses similarly to other common kinds of data, we used Mpigz to compress three other types of data in addition to our Ubuntu image.

An executable file workload was created by extracting all ELF files from the Ubuntu image (as identified by the `file` utility), resulting in a text consisting of only executable binaries and shared libraries of AMD64 code. An uncompressed tar file containing the Linux Kernel 2.6.39.3 source served as a source code workload. Finally, we acknowledge many real-world data types that may be encountered by transparent network compression compress poorly. We wanted to include some difficult-to-compress data (that was not already compressed or encrypted) to exercise this case. PCM audio is generally unsuitable for the LZ77 and Huffman coding used by deflate and is thus somewhat of a "torture test." Accordingly, we used a tar file containing the uncompressed, 24-bit, 96KHz PCM (pulse coded modulation) audio from the album *The Slip* by Nine Inch Nails, freely distributed under the Creative Commons Attribution Non-Commercial Share Alike license.

Figure 2 shows that the Ubuntu image, Ubuntu-binaries, and Linux Kernel source curves scale with a similar shape as compression strength is increased. This suggests that our standard workload scales comparably to other kinds of realistic compressible data, even though the texts are of different sizes and achieve different amounts of absolute compression. However, even though they compress reasonably well, we see that increasing the strength parameter from 1 to 9 results in only 6-7.2% more compression at the expense of 350-1000% more time. In other words, strength 1 compression has the highest reduction rate for these three cases — almost all of the compression already occurs at the lowest strength, but at a much lower CPU cost. This is not an artifact of Mpigz or zlib; a similar effect occurred with GNU Gzip in Figure 1, where switching from strength 1 to 9 provided only 6.4% more compression at the cost of over 800% more time. This confirms that zlib and GNU Gzip, two of the most common deflate implementations, appear to scale quite poorly in terms of reduction rate as compression strength increases.

In contrast, the poorly-compressible PCM data doesn't exhibit this curve. While compression time does increase with strength level, strength 9 is only about 5% slower than strength 1. Additionally, no strength level achieves greater than 1.90% total reduction. Two clusters form; strengths 1 though 3 perform similarly, then there is a gap, and strengths 4 through 9 perform similarly. Strangely, strength 2 actually achieves greater reduction than strength 3. Without overgeneralizing, this suggests that while legacy strength settings make a small (but extremely costly) difference for compressible data, they may be even less helpful for difficult to compress data.

Surprisingly, the default compression strength for both zlib and GNU Gzip is 6. This may be suitable for off-line, persistent file compression, as it appears to be near the inflection point for compressible data. However, when compressing our standard image, strength 6 achieves a reduction rate of about 10 megabytes per second, while strength 1 achieves more than double — over 22 megabytes per second. This general trend exists for all our compressible data and suggests that strength 1 may be more appropriate when it is essential that compression not become a bottleneck (as in OpenSSH [1], where compression is an off-by-default option), because strength 1 is much faster than strength 6 with only a minor decrease in compression. This would also free up CPU cycles for more valuable uses, such as managing transmission. In fact, some sources have claimed that over fast networks (like local Ethernet), strength 1 is always a better option than the default strength 6, because the reduced CPU usage itself leads to a lower overall transmission time[20].

3) Bridging the Gap with Special Compression Modes: In addition to the limited scaling provided by the strength parameter, there is a huge performance gap between strength 0 and strength 1 compression. This is visible in Figure 3 where (on our standard workload) strength 1 achieves 94% of the maximum reduction in about 50 seconds — less than half the time of the default and roughly one-ninth the time of strength 9. Unfortunately, there is no way to achieve less or faster compression using standard strengths. As a result, when where there is not enough CPU time to use strength 1, simple switching schemes have no option but to send raw data — even if they could have afforded to sacrifice a small amount of CPU time in exchange for some small degree of compression. This illustrates the potential value of scaling between strengths 0 and 1.

One possibility is that the specialized, lightweight compression modes included in zlib (Filtered, RLE, and

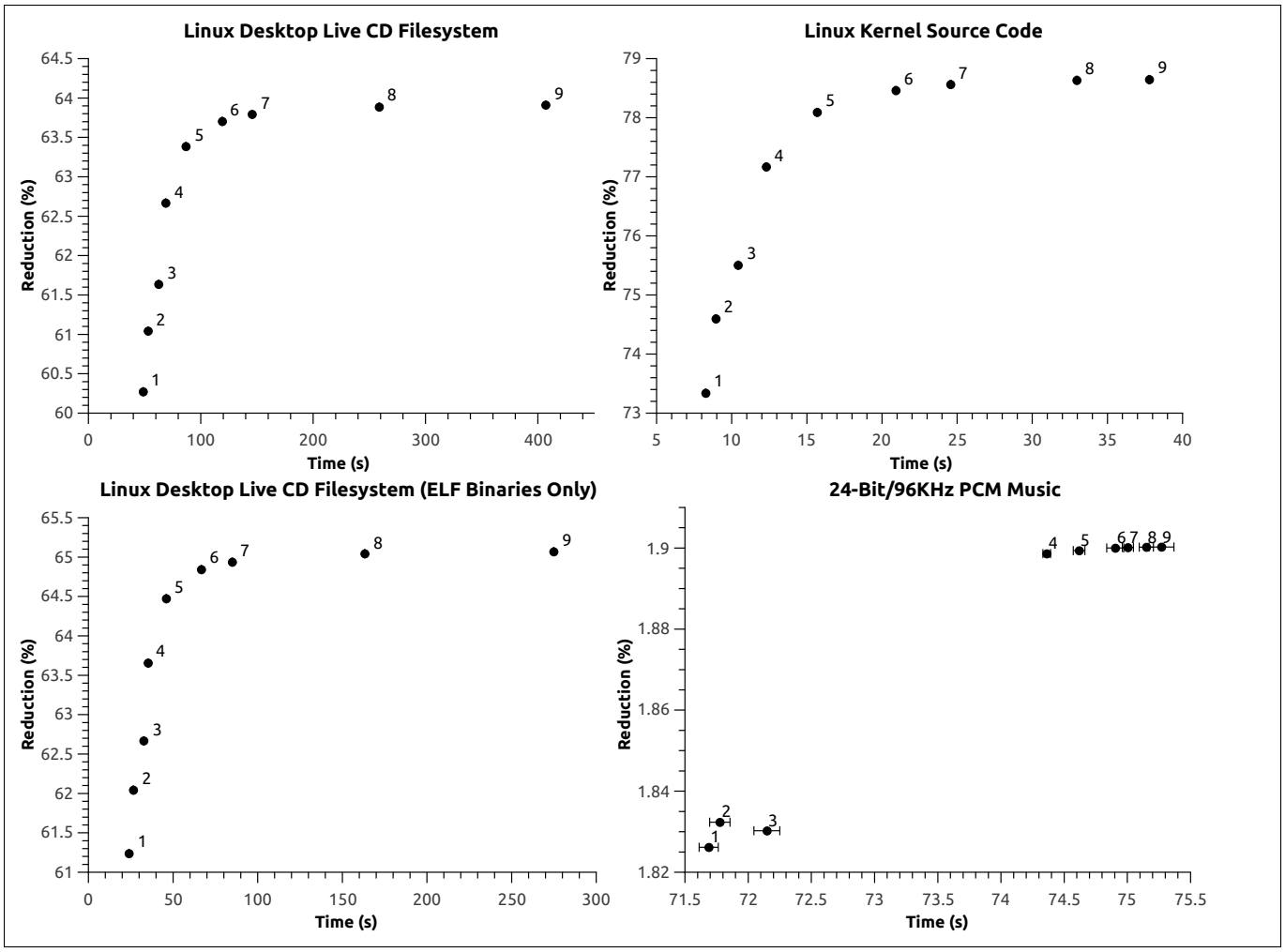

Figure 2. Comparison of the scaling behavior of different data types. Labels represent the strength parameter passed to the compressor. (Higher represents more compression.)

Huffman-Only)[2] could serve this purpose. Figure 3 shows the standard zlib strengths 0-9 and the lightweight modes, in addition to our own direct generation (DG) code. In these tests, Filtered is not helpful for our purposes, achieving less reduction than strength 6 in roughly the same amount of time. In contrast, HO and RLE are approximately 28% faster than strength 1 compression, but this speed comes at a 41% expense in compression. While HO or RLE could be useful as a "strength 0.5," albeit with lower efficiency (in terms of reduction rate), there would still be large performance gaps on both sides where no option exists. Additionally, because these special compression modes run the risk of being highly data-dependent [21], they may be unreliable in practice.

4) Simple Mixing and Opportunistic Compression: Next, we investigated the use of fine-grain mixing to bridge the "compression gap." Figure 4 shows three data series: the standard zlib strengths 1 through 7 (8 and 9 were omitted for space but are visible in Figure 3), directly generated blocks (DG) mixed with strength 1 (the most efficient in terms of reduction rate), and DG mixed with default strength 6 blocks as used by Pu & Singaravelu.

In Figure 4, we see fine-grain mixing of compression scaling down from the standard compression strengths linearly with decreasing CPU usage. In other words, the gap in the compression curve between no compression and strength 1 compression has been effectively bridged with a range of mixing percentages. Regression lines are plotted to underscore the linear scaling provided by fine grained mixing. The linear regressions for strength 1 and 6 mixing have $R^2 = 0.999967$ and $R^2 = 0.999987$ respectively, indicating extremely linear scaling in terms of reduction versus compression time.

This linearity shows clearly that — in addition to its

[2]As described in Section III-B, zlib 1.2.4 or newer will be required to duplicate these results due to slow performance of the lightweight modes in earlier versions.

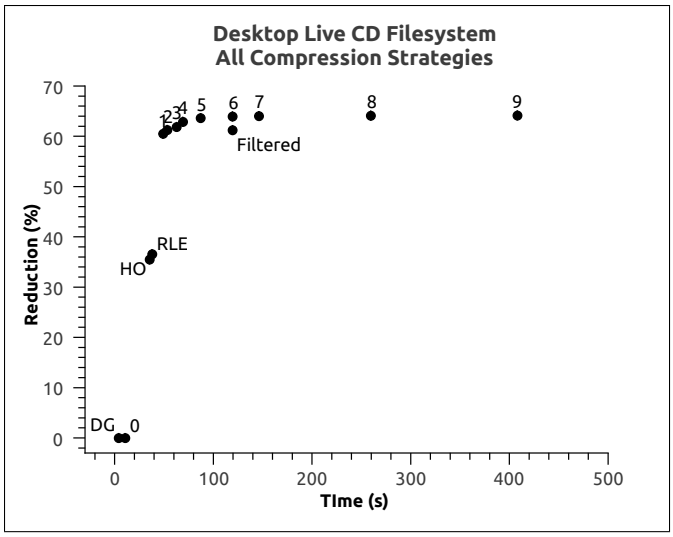

Figure 3. Comparing all non-mixed compression strengths.

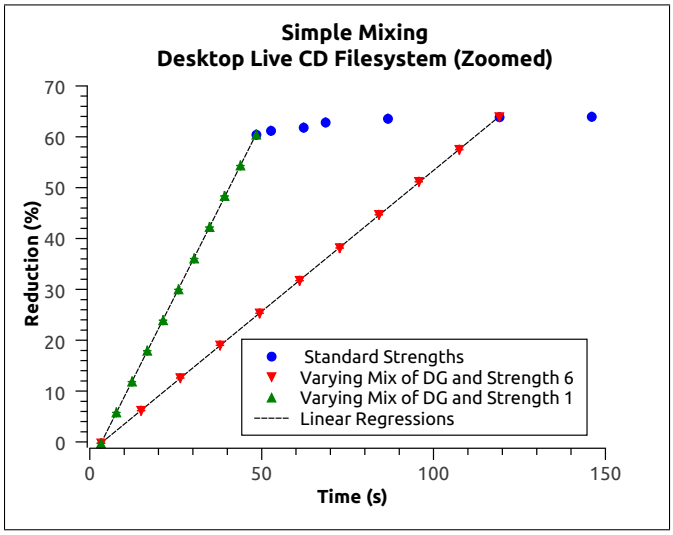

Figure 4. Simple mixing of compression strengths with linear regressions. Standard strengths 8 and 9 omitted for space.

ability to maximize network and CPU utilization — FG-Mixing provides extremely responsive yet efficient control over the tension between data reduction and time, with a much wider performance spectrum than the nine standard compression strengths.[3] Additionally, it is clear that while the *maximum* reduction of strength 1 is less than that of strength 6, mixing with strength 1 outperforms mixing with strength 6 in terms of reduction rate. In other words, mixing with strength 1 increases compression faster than strength 6 with respect to CPU time. Thus, it would be more efficient to mix with strength 1 up to its maximum compression, and then mix with strength 6, rather than to initially mix with strength 6. (Since Pu and Singaravelu's original experiments focused on resource utilization and didn't directly investigate scaling, they mixed with the default strength 6 blocks rather than the more efficient strength 1 blocks. This is one way in which the understanding of FG-Mixing's scaling behavior can result in increased performance.)

In previous adaptive compression systems, whether and how to compress had to be efficiently predicted based on current conditions. However, Figure 3 shows that mispredictions (such as choosing to compress while "in the gap") are costly, and even when predictions are correct, increasing the legacy strength level provides a very small benefit at significant CPU cost. In contrast, FG-Mixing has to make no such decisions up front; it can simply compress blocks until more compression would block transmission. Compressing a greater percentage of blocks simply causes a shift to the right on the linear mixing region of the curve, with the resulting amount of compression and CPU time increasing by the same amount for each additional block compressed. Thus, not only is there no cost or risk of a prediction scheme, but the outcome of increasing or decreasing the mixing percentage is reliable and consistent.

5) Scaling Up with More CPU Time: From Figure 4 and the previous discussion, it seems likely that many applications could be well-served by mixing of strength 1 only — using 100% strength 1 compression as their strongest and most costly strategy. In our tests, unmixed strength 1 compression provides almost all the compression benefit at the lowest CPU cost. Additionally, in our tests on compressible data, increasing the proportion of strength 1 to DG blocks always improves reduction and does so more quickly than mixing with higher strengths (as long as the reduction is below the maximum achievable by strength 1). However, in scenarios where trading small increases in compression for significant amounts of CPU is valuable, higher compression strengths can be utilized as well. In fact, different non-zero strength blocks can even be mixed together to provide an extremely fine-grain range of choices up to the limits of a particular compression implementation.

In Figure 5, we mix from DG blocks up to strength 1 blocks in 5% steps. After reaching 100% strength 1 blocks, we progress through strengths 1 to 9 in 50% steps, demonstrating mixing between higher compression strengths. Adaptively choosing a point in the linear region between DG and the strength 1 point is simple; blocks can simply be compressed opportunistically and emitted as DG blocks if not compressed in time. Choosing between the higher-strength points requires some sort of heuristic, but as long as it quickly falls back into the predictable and opportunistically-mixed DG to strength 1 region, the specific heuristic is less critical than in adaptive compression systems with less granularity.

[3]It is worth noting that unlike zlib's deflate, some compressors do not have strength levels at all, or if they do they may be *less* effective. However, the basic concept of FG-Mixing should work with many other compressors, as the original work explores.

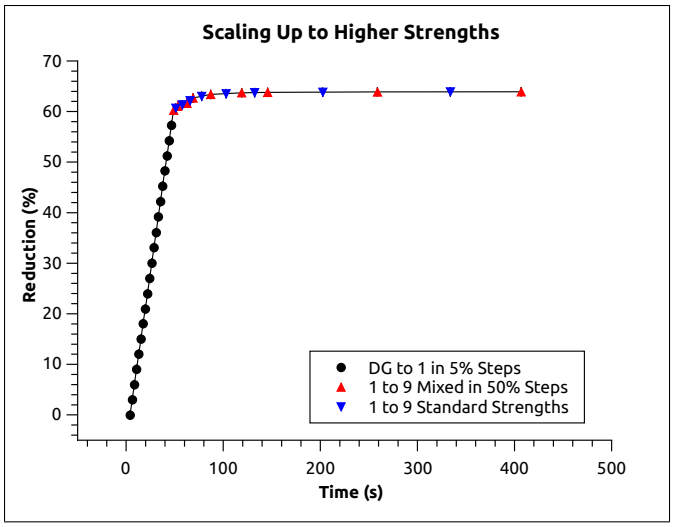

Figure 5. Mixing Higher-Strength Compression to Scale Up with More CPU.

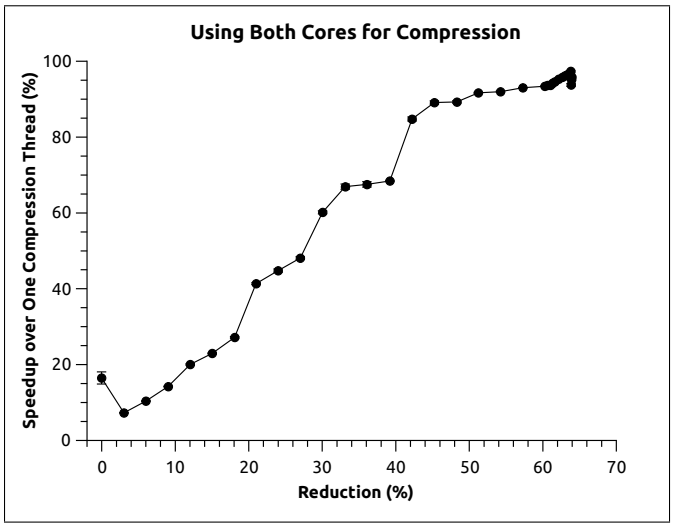

Figure 6. Speedup of two-threaded mixed compression versus single-threaded mixing.

6) Scaling Up with More CPU Cores: Of course, one obvious route to speed up compression would be to utilize more CPU cores. (This was the original purpose of pigz's block-based architecture.) Pu & Singaravelu tested a dual-thread implementation, but they used separate compression and transmission threads, rather than multiple compression threads. In Figure 6 we plot the speedup attained by employing a second compression thread on our dual-core test machine as the amount of reduction is varied when compressing our standard workload. Speedup was calculated using real times, as opposed to user times, to now include rather than exclude the effect of multiple CPU cores. A speedup of 100% would correspond to a doubling of execution speed due to the second core.

Employing multiple compression threads dramatically reduces compression time, and this effect increases as a greater percentage of blocks are compressed. Close to the origin, direct generation sees very little speedup (likely due to DG's dependence on the throughput of `memcpy`). However, the speedup climbs towards 95% as the mixing approaches 100% strength-1 compression. At the tail end of the plot, consisting of mixed higher-strength strategies, we approach close to an ideal 2x speedup before dropping slightly at the highest mixing levels.

7) Maximum Reduction by Output Rate: Another way to measure the performance of an adaptive compression scheme is to look at the maximum achievable compression for a given output rate. Since this rate often corresponds to available network bandwidth, this addresses the question "if I must send data at rate X, how much gain can compression provide before it becomes the bottleneck?"

Figure 7 plots the answer with our standard single-threaded setup on a logarithmic X-axis. The highest output rate, 673 megabytes per second (approximately 5.5 gigabits per second), is the direct generation rate. In most real-world networking scenarios, there will be much less available bandwidth. Significantly, the majority of the curve represents mixing with strength 1. With maximum compression (unmixed strength 9), the highest output rate is only about 1.6 megabytes per second, comparable to a wireless LAN.

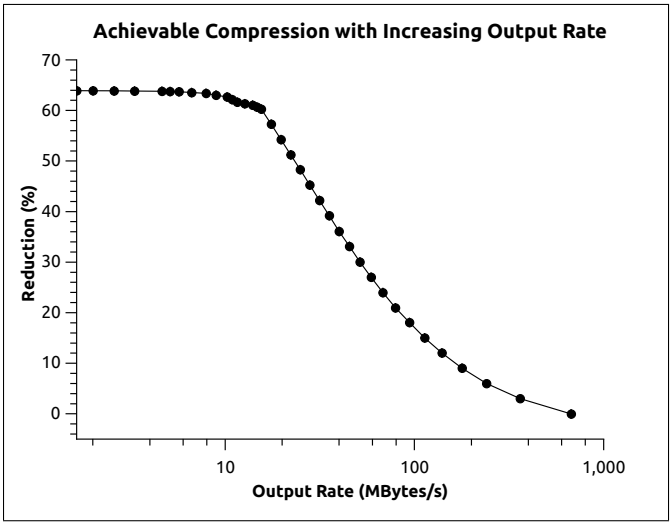

Figure 7. Compression achieved with increasing output rate in megabytes per second of real time. Logarithmic scale shows detail of the lower output rate region.

8) Effects of Block Size and Mode on Scaling Behavior: In addition to the strength parameter, the input block size and block history sharing modes can significantly affect the reduction and runtime of FG-Mixing. As described in Section III-A, "independent" mode is typically faster because it compresses each block in isolation, while "shared history" mode often achieves more compression by initializing each

block's compressor with the history of the previous block. The differences in runtime and reduction between the two modes are expected to diminish with increasing input block size, because the per-block cost of history sharing will affect fewer [larger] blocks. Simultaneously, the penalty of independent mode on reduction is expected to diminish as blocks grow larger, because the fixed-size history window will be under-filled for a comparatively smaller fraction of each block. To explore these effects, we test multiple block sizes with independent compression both enabled and disabled.

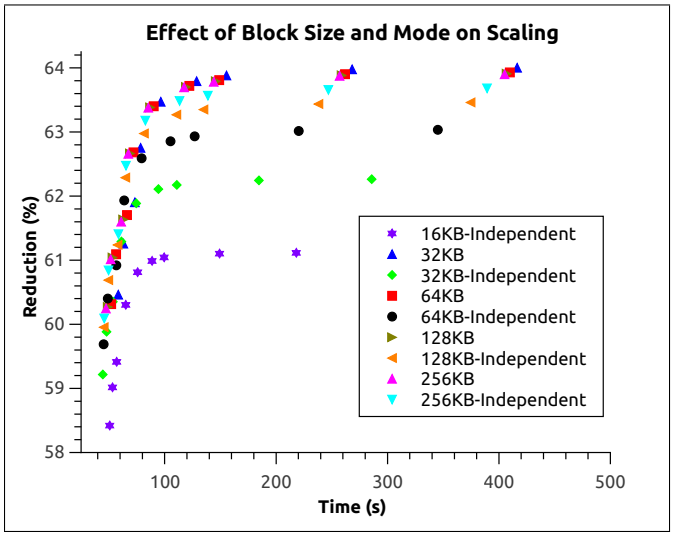

Figure 8. Varying block sizes and modes. Strength increases from 1 to 9 from left to right. Error bars are not shown to avoid clutter, but standard deviations are similar to previous tests.

The results are shown in Figure 8. Several patterns can be observed. First, the 16KB independently compressed blocks as used by Pu and Singaravelu[8][4] are sub-optimal in terms of reduction rate and maximum compression in our tests. While small block sizes are attractive in that they offer more opportunities for adaptation, in all cases another block size can provide greater compression for the same CPU cost. Moreover, the maximum reduction of 16KB independent mode is about 4% below the best strategy.

Another observation is that "shared history" mode always compresses better than "independent" mode, but this effect decreases as block size increases. With 32KB blocks, the difference in reduction between the two modes is approximately 2%, while with 256KB blocks, the difference is only about 0.2%. On the other hand, "independent" mode is always faster relative to an equivalent "shared history" configuration, but this effect also diminishes as input size increases. With 32KB blocks at strength 9, independent mode finishes about 25% faster than shared mode, but with 256KB blocks, independent mode is only around 5% faster.

"Shared history" inhabits the top left region of Figure 8, indicating greater efficiency, and doesn't "top out" at smaller block sizes like "independent mode". This tolerance of smaller input blocks can be quite useful in dynamic situations. Smaller blocks allow shorter commitments to a particular mixing percentage, enabling faster adaptation to shifting conditions. "Shared history" mode thus provides greater efficiency and adaptivity, and seems more suitable for opportunistic adaptive compression.

V. Future Work

There are many ways that the performance of Mpigz could be improved. First, it is possible that the compressor itself could be optimized, although zlib is mature. It seems more likely that our DG routine could be improved. Our routine is currently limited by the throughput of `memcpy` (that is, the memory bandwidth of the machine), but techniques such as scatter-gather and zero-copy I/O, hardware-assisted CRC generation, or the utilization Non-Uniform Memory Access (NUMA) architectures may allow better DG throughput. Finally, it may useful to examine system-wide performance when compressing multiple streams; if one stream consists of poorly-compressing data, it may be beneficial to switch to a lower mixing ratio and "donate" CPU time to a better-compressing stream.

VI. Conclusion

The compression strength options of common compressors like zlib and GNU Gzip confine applications to a handful of coarse-grain choices in terms of CPU consumption, but with extremely similar performance in terms of compression ratio. This can lead to resource underutilization, even when adaptive compression schemes make the right strategic choice, because no legacy option perfectly fits the scenario. Additionally, the compression performance gap between no compression and any compression (common in legacy compressors) limits adaptive compression schemes to situations that admit a significant investment in computation.

Pu & Singaravelu's work on FG-Mixing was originally designed as a technique to maximize the utilization of both CPU and bandwidth simultaneously. Instead, we explored FG-Mixing as a technique for scaling legacy compression, showing how mixing compressed and uncompressed blocks provides legacy compressors with a wide spectrum of efficient choices, unlike those provided by compressor strength options. This effectively bridges the "compression gap," making compression much more useful for throughput-sensitive applications such as network transmission.

This focus led to several improvements over the original work. While Pu & Singaravelu's implementation used 16KB blocks compressed independently at the default strength, we

[4]The default block size and mode for Pu and Singaravelu's implementation were obtained from their source code and private correspondence, although one explicit test in their paper varies the block size from 8KB to 64KB.

showed that we could achieve more compression with significantly less computation by compressing somewhat larger input blocks with strength 1 compression and shared block histories. We found that strength 1 compression achieves virtually all the benefit of strength 9 for a fraction of the CPU cost. Nevertheless, if excess CPU remains even after compressing every block at strength 1, fine-grain mixing between higher strengths in arbitrary ratios enables an extremely fine granularity in the trade-off of compression versus computation, up to the limits of the algorithm itself.

In addition to increasing efficiency, we've also optimized for high-speed networks while retaining backwards-compatibility with deflate. Our code generates uncompressed blocks much faster than zlib, allowing our single threaded implementation to scale up to 673 megabytes per second (5.5 gigabits per second) with a single thread, faster than most local area networks. Our implementation also supports parallel compression, almost doubling throughput when compressing heavily. Moreover, full backwards compatibility with the deflate specification means that legacy deflate decompressors, installed worldwide, can process our streams without modifications. This virtually eliminates the barrier to adoption and allows for incremental rollout in existing, heterogeneous, and widely distributed systems.

In sum, we believe our analysis of FG-Mixing and resulting improvements bring adaptive compression closer to being a viable on-by-default technique for improving the performance of existing and future distributed, network, and mobile systems.

VII. Acknowledgments

We gratefully acknowledge Calton Pu and Lenin Singaravelu for their work, source code, and correspondence.

References

[1] *SSH(1)-The OpenBSD Reference Manual*, August 2011, version 4.8. [Online]. Available: http://www.openbsd.org/cgi-bin/man.cgi?query=ssh

[2] C. Krintz and S. Sucu, "Adaptive on-the-fly compression," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, pp. 15–24, 2006.

[3] E. Jeannot, B. Knutsson, and M. Björkman, "Adaptive online data compression," in *IEEE International Symposium on High Performance Distributed Computing*, 2002, pp. 379–388.

[4] B. Knutsson and M. Björkman, "Adaptive end-to-end compression for variable-bandwidth communication," *Computer Networks*, vol. 31, no. 7, pp. 767 – 779, 1999.

[5] Y. Wiseman, K. Schwan, and P. Widener, "Efficient end to end data exchange using configurable compression," *SIGOPS Oper. Syst. Rev.*, vol. 39, pp. 4–23, July 2005.

[6] P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," RFC 1951 (Informational), Internet Engineering Task Force, May 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1951.txt

[7] K. G. Morse, "Compression tools compared," *Linux Journal*, vol. Issue 137, 2005. [Online]. Available: http://www.linuxjournal.com/node/8051/print

[8] C. Pu and L. Singaravelu, "Fine-grain adaptive compression in dynamically variable networks," in *International Conference on Distributed Computing Systems*, 2005, pp. 685–694.

[9] P. Deutsch and J.-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3," RFC 1950 (Informational), Internet Engineering Task Force, May 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1950.txt

[10] P. Deutsch, "GZIP file format specification version 4.3," RFC 1952 (Informational), Internet Engineering Task Force, May 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1952.txt

[11] (2007, September) .zip file format specification. Version 6.3.2. [Online]. Available: http://www.pkware.com/documents/casestudies/APPNOTE.TXT

[12] T. Pornin, "Zlib flush modes," August 2007. [Online]. Available: http://www.bolet.org/~pornin/deflate-flush.html

[13] A. Tridgell, "Efficient algorithms for sorting and synchronization," Ph.D. dissertation, Australian National University, 1999.

[14] *Portable Network Graphics (PNG) Specification (Second Edition)*, ISO/IEC Std., November 2003. [Online]. Available: http://www.libpng.org/pub/png/spec/iso/index-object.html

[15] "Aha367-pcie 10.0 gbits/sec gzip compression/decompression accelerator." [Online]. Available: http://www.aha.com/

[16] D. A. Huffman, "A method for the construction of minimum redundancy codes," 1962.

[17] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[18] M. Adler, "pigz - parallel gzip," January 2010. [Online]. Available: http://www.zlib.net/pigz/

[19] *GZIP(1)-FreeBSD General Commands Manual*, April 2010, version 8.2. [Online]. Available: http://www.freebsd.org/cgi/man.cgi?query=gzip

[20] D. J. Barrett and R. E. Silverman, *SSH, The Secure Shell: The Definitive Guide*. O'Reilly & Associates, 2002, section 7.4.11.

[21] J. loup Gailly and M. Adler, *zlib 1.2.5 manual*, April 2010. [Online]. Available: http://www.zlib.net/manual.html