Analysis of Web Data Compression and its Impact on Traffic and Energy Consumption

Frank Ebner and Volker Schneider
University of Applied Science Würzburg-Schweinfurt
Würzburg

While connection speeds are increasing slowly, some ISPs mention plans about possible traffic limitations in the near future which would keep internet traffic expensive. In addition, Green IT became more important especially over the last few years. Besides on-demand content like video live streams, HTTP traffic plays an important role. Thinking of textual web content, compression quickly comes to mind as a possibility to reduce traffic. The current HTTP/1.1 standard only provides gzip as an option for content encoding. HTTP/2.0 is under heavy development and numerous new algorithms have been established over the last few years. This paper analyzes HTTP traffic composition on a production server and concludes that about 50% is compressible. It further examines the effectiveness of custom and existing algorithms with regards to compression ratio, speed, and energy consumption. Our results show that gzip is a sound choice for web traffic but alternatives like LZ4 are faster and provide competitive compression ratios.

Categories and Subject Descriptors: E.4 [Coding And Information Theory]: Data Compaction and Compression; D.4.6 [Computer-Communication Networks]: Network Operations—*Network Monitoring*

Additional Key Words and Phrases: Web, HTTP, Compression, Energy, Traffic

1 INTRODUCTION

Internetworking of various systems plays an increasing role in everyday life. The demand for bandwidth is growing steadily because more and more devices depend on network access but faster up- and downlinks are costly. At the same time, the current trend towards Green IT requires devices to manage their power consumption more efficiently. Data compression could be part of the solution for limited bandwidth and increased energy efficiency. The current HTTP/1.1 standard allows only gzip for body compression [Fielding et al. 1999]. We will analyze the consequences of this constraint and compare the compression ratio, speed, and energy consumption of gzip and other algorithms using real-life HTTP traffic.

Paper organization: The following document is organized into three consecutive sections, each of which presents its own results and conclusion. Section 2 analyzes HTTP traffic composition of a medium-size company's server to get an impression of its compressibility. Section 3 examines the compression ratio and speed of selected algorithms for the content-types that were analyzed previously. In addition, the potential of static Huffman and dictionary coding is determined. Section 4 presents the setup and results of energy tests analyzing the impact of compression on the server's and the client's energy consumption. Section 5 presents an outlook for future work.

Author's address: Frank Ebner, Volker Schneider; Fakultät für Informatik und Wirtschaftsinformatik, Hochschule für Angewandte Wissenschaften, Sanderheinrichsleitenweg 20, Würzburg, Deutschland

2 TRAFFIC ANALYSIS

To get an impression of HTTP traffic compressibility the first step is to analyze the average payload's composition on a production server. As video and download portals mainly use already compressed traffic and their amount of compressible contents are negligible, we will analyze a server hosting basic websites only, with a mixture of static and dynamic payload of html, xml, css, js, json, images and other. Common image formats, like jpeg and png, already use ideal compression algorithms and are thus not further compressible. Hence, the amount of html, xml, css, js, json and similar based payloads correlates with the traffic's compressibility. We will analyze live HTTP traffic provided by a production web server of a medium-size company and track several traffic-stats like latency, distribution, and compressibility. The payloads are exported to test the impact of chunk sizes and compression levels using another machine.

The analysis was performed on the web server M_{s1} which is part of the production environment hosting German and English websites using PHP (see Table 1). All data sent from the server upon request was captured and exported to disk for later analysis. Confidentiality of the data was ensured at all times. The server's traffic was captured for about 72 hours and referred to as P_{72}.

Machine	CPU	RAM
M_{s1}	Intel® Core™ i7-3770	16 GB DDR3
M_{a1}	Intel® Core™ i3-M380	4 GB DDR3 1066

Table 1: Used machines for live analysis and further tests

2.1 Software

For traffic analysis and compression tests we developed NetAnalyzer, a tool to capture HTTP traffic and examine various important aspects. To provide an easy integration into production environments, we decided against writing a HTTP proxy. Instead we used libpcap to capture all TCP packets on port 80 and extract the actual HTTP streams after IPv4 and TCP reassembly. This ensures a very simple integration at the cost a slight packet loss / skipped connections in some environments (about 1% in our case) depending on the used hardware [Papadogiannakis et al. 2012]. To compensate for the latter and to ensure plausible results, tests and exports were repeatedly run over a period of several days.

The captured packets traverse IPv4 reassembly and are piped into a TCP reassembler which assigns them to connections using both IPs and ports [Postel 1981]. Later, the TCP streams are analyzed by the HTTP decoder to split header / payload and distinguish between requests and responses. HTTP data is then streamed through several configurable compressors and analyzers to track e.g. the web server's latency, the time needed for compression, and the resulting compression ratio for each content-type. The gathered statistics and the captured payloads are exported for further evaluation (section 2.2) and to serve as representative test data for the energy tests in section 4. The HTTP decoder triggers the analyzers directly at packet level without additional buffering. Hence, the compressors will append data blocks of about the average MSS which is, at max, 1460 bytes when using Ethernet with a MTU of 1500 bytes [Postel 1983]. As compressors like gzip are configured to use `Z_NO_FLUSH`, they are independent of this chunk size. Other compression algorithms like LZO and LZ4 however are strictly block based, their compression ratio might vary significantly depending on the block size (see section 3).

Equally important as the traffic's composition are the latency distributions. As disk access is much slower than the overhead needed for compression [Yang et al. 2010], the CPU time spent on compaction of uncacheable files, like dynamic websites using PHP, Ruby, Python, or other languages, is less critical than compressing static websites that will most likely reside within the file system cache. Therefore, we try to analyze the amount of (compression-suited) dynamic content by exporting latency values and applying an appropriate threshold.

2.2 Results

2.2.1 Traffic distribution

The analyzed production server's HTTP request payloads (24.6 MiB) had about 2‰ the size of the response payloads (12.9 GiB) and thus are statistically insignificant. Therefore, we did not further analyze the compressibility of the requesting side. Table 2 provides the detailed results of the response payload's composition. We also did not examine the HTTP headers' compressibility as the NetAnalyzer tool reported an average header size of only 561 bytes for requests and 319 bytes for responses, respectively. Furthermore, HTTP/1.1 currently does not allow such a construction. The upcoming HTTP/2.0, however, will provide multiplexing with corresponding header-compression [Belshe et al. 2013][Peon and Ruellan 2013].

	html	images	js	css	xml	json	other
traffic	5.54 GiB	5.35 GiB	1.37 GiB	0.36 GiB	0.04 GiB	0.02 GiB	0.21 GiB
traffic %	42.9%	41.4%	10.8%	2.8%	0.2%	0.1%	1.8%
files	113,737	385,927	67,138	39,498	4,557	6,212	8,689

Table 2: Traffic distribution within P_{72}

Our expectation of the used image formats was confirmed as we mainly found jpeg images (4.59 GiB / 233,858 files) followed by png (0.67 GiB / 77,382 files) and gif (0.09 GiB / 74,687 files). Those formats already have a high entropy of nearly 8-bit per symbol (byte) and thus are not suitable for further compression. All following tests will use a 24 hour slice P_{24} of P_{72}. Unfortunately, most of the xml traffic was occupied by two files, requested every minute. After removing those files, to ensure a variety of different payloads, P_{24} contained only 25 xml files with 500 KiB in size. Therefore, xml tests use the larger window of P_{72} containing 68 files using 2.7 MiB. This amount still is negligible compared to other content-types and xml test results should be handled with care. Figure 1 shows the traffic distribution on the web server M_{s1} within the extracted range of P_{24}. The distribution on the analyzed web server strongly depends on the time of day and allows higher compression at nightly hours. The average traffic of one day consists of about 50% well suited for compression. Table 3 and Figure 1 depict the detailed traffic composition.

	html	js	css	json	xml
traffic	2,033.1 MiB	550.9 MiB	148.7 MiB	6.6 MiB	2.6 MiB
traffic %	74.1%	20.1%	5.4%	0.2%	0.1%
files	33,570	26,444	15,264	1,979	68
avg filesize	62.1 KiB	21.3 KiB	10.0 KiB	3.4 KiB	39.1 KiB

Table 3: Extracted content-types within P_{24}

2.2.2 Traffic latency

The left half of Figure 2 shows the histogram for html content latencies of P_{72} between 0 and 1000 milliseconds. 108761 values matched this latency range which is about 95.6% of the complete payload P_{72} (see Table 3). $Q_{25} = 6$ claims that many responses were delivered almost instantly. The right half shows the histogram for low latencies between 0 and 15 milliseconds. Starting at 8 milliseconds, the number of entries per bin almost seems to be stable. Therefore, we decided to split between static and dynamic contents using a threshold of 10 milliseconds. Table 4 depicts the results, using this threshold for P_{24}.

	html	js	css	json	xml
num static	25.3%	96.2%	94.0%	3.5%	2.0%
size static	12.8%	93.0%	87.3%	7.6%	1.2%

Table 4: Static / Dynamic distribution per content-type within P_{24}

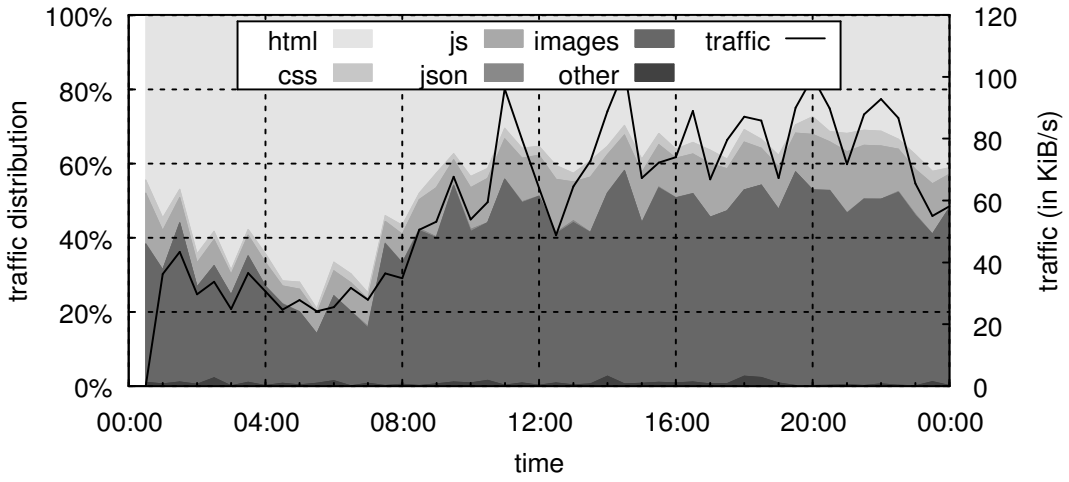

Figure 1: Distribution and amount of HTTP traffic (30 minutes average) on M_{s1}

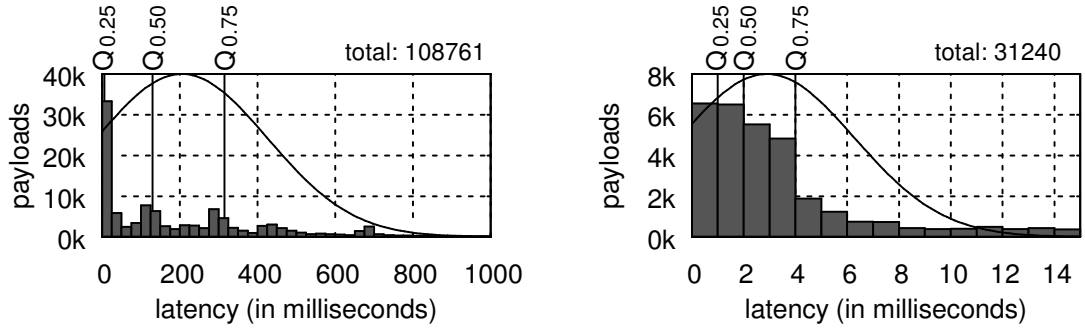

Figure 2: Response latencies for html payloads within P_{72}

2.3 Conclusion

On the tested server M_{s1} the amount of compressible payload is in fact considerable. It is composed of html, js, and css which account for about 56.5% of the total traffic. In addition, only one tenth of the html traffic is based on static content indicating a negligible overhead for compressing a significant amount of payload.

3 COMPRESSION ANALYSIS

In this section we select five existing compression libraries and evaluate them using the web traffic P_{24} from section 2.2. In addition to the existing compression libraries we examine the effectiveness of two custom compression methods in section 3.2 and 3.3. The results serve as input for choosing the most appropriate compression algorithms for the energy tests in section 4.

Before presenting the compression libraries we need some definitions that will be relevant throughout this section. We define the terms compression ratio ΔN, compression speed S_c, and reduction speed S_r as in Equation 1 where t_c is the time the compressor C needed for compressing N uncompressed bytes to n compressed bytes.

$$\Delta N = \frac{N}{n} \qquad S_c = \frac{N}{t_c} \qquad S_r = \frac{N-n}{t_c} \qquad (1)$$

Compressibility depends on the input's entropy (see Equation 2) which is about the average number of bits needed to describe one of its code words (e.g. one byte, one word, one sentence) [Blelloch]. Most compression algorithms thus are based on either a form of entropy encoding, like Huffman coding [Huffman 1952], and / or some kind of dictionary, like those of the Lempel-Ziv family [Ziv and Lempel 1977]. As these compressors have no a priori knowledge of the to-be-compressed payloads they usually are adaptive to provide good compression ratio. However the adaptive behavior is CPU intensive as the data structures which provide high searching speeds must constantly be updated e.g. when gzip's sliding window changes [Bell and Kulp 1993]. Thus we will take a closer look at a static entropy encoder / dictionary to check the feasibility of static algorithms using a priori knowledge and analyze obtainable compression ratios.

$$H(X) = \sum_{i=1}^{n} p(x_i)\, log_2\, (\frac{1}{p(x_i)}) = \sum_{i=1}^{n} -p(x_i)\, log_2\, p(x_i) \qquad (2)$$

3.1 General Purpose Algorithms

This section presents the five compression libraries chosen for the initial payload analysis. Each library is referred to as "algorithm" for better readability, although we chose a concrete implementation with a specific version and configuration. All algorithms are lossless and intended for general purpose usage. Overall the choice was biased towards algorithms known for fast compression and decompression speed.

3.1.1 gzip

The GNU zip (gzip) is a combination of the GNU zip file format and data compressed with DEFLATE algorithm. It can be created using the `gzip` program or the zlib library. gzip is allowed as an encoding format by the HTTP/1.1 standard [Fielding et al. 1999] and defined in [Deutsch 1996b]. The current RFC draft for the upcoming HTTP/2.0 standard [Belshe et al. 2013] does not add new compressed encoding formats, although there have been attempts [Butler et al. 2008] to introduce new formats into the current standard in the past. gzip serves as a reference regarding compression and speed. It supports different compression levels from 1 (fastest) to 9 (best ratio) and can use huffman-only compression strategy. We will refer to different configurations of gzip: $gzip_1$, the fastest compression level; $gzip_6$, the default compression level; $gzip_9$, the level providing the best compression ratio; $gzip_{huffman}$, using only Huffman coding. HTTP/1.1 allows the usage of deflate content encoding, also known as zlib format [Deutsch 1996c] with DEFLATE compression, in addition to gzip. We will focus on gzip since deflate encoding is reported to be the more reliable choice across different servers and browsers.

3.1.2 LZMA

Lempel–Ziv–Markov chain algorithm (LZMA) is a compression algorithm that uses a dictionary compression scheme. It is known for its high compression ratio for most contents. LZMA is used as the default compression method of the 7z format. The reference implementation was placed in the public domain in 2008. Although the API provides options to customize the compression and decompression experience (level 1-9), the usage of default parameters leads to high memory requirements especially for compression. The manual page for the `xz` program gives a rough estimate of the memory requirement for level 1 as 9 MiB for compression and 2 MiB for decompression. Level 4 would require about 48 MiB for compression and 5 MiB for decompression. The analysis of the web traffic focused on using LZMA at level 1 with a CRC64 checksum. Default values were used for all other arguments. This configuration is hereafter referred to as $lzma_1$. We chose LZMA to have a reference with regards to compressibility of the traffic data.

3.1.3 LZO

Lempel–Ziv–Oberhumer (LZO) is a compression algorithm that focuses on fast decompression. Its default algorithm requires only 64 KiB memory for compression. The compression is known to be fast while favoring speed over a higher compression ratio. LZO is licensed under the GPLv2+. Its area of application includes compression in the Btrfs file system, optional data stream compression in OpenVPN, and Linux Kernel compression. The API provides algorithms of different flavors. The traffic analysis used the LZO1X-1 algorithm as recommended by the official LZO FAQ when aiming for speed. This configuration is hereafter referred to as lzo_1. Initial tests were also conducted using LZO1X-999 but failed to provide significantly better compression ratio.

3.1.4 LZ4

LZ4 is an algorithm for fast compression and decompression. It offers two kinds of API, default and high compression mode (HC). The traffic analysis used LZ4 r94 in its default configuration hereafter referred to as lz4. The algorithm's source code is published under the 2-clause BSD license. Although the project is relatively young, it has seen wide adoption lately. It is actively used in GRUB, the ZFS file system and is supported for Kernel compression since Linux 3.11.

3.1.5 QuickLZ

QuickLZ is another fast compression algorithm. It is licensed under the GPL but also offers commercial licensing. The algorithm provides three compression levels that allow the user to choose between compression or decompression speed. In addition, it offers the usage of a history buffer for improved compression ratio and an option for memory safe decompression for corrupt input data (15-20% slower). The analysis was performed with library version 1.5.0 configured with level 1 (fastest compression speed), a streaming buffer of 100000 bytes and no extra memory safety. This configuration is referred to as $quicklz_1$ for the rest of this document.

In addition to the general purpose algorithms we examined the feasibility of non-adaptive compression using one representative of dictionary and entropy coding, respectively. For the latter, we decided to settle for the well-known Huffman coding.

3.2 Huffman coding

Our implementation builds the necessary (static) Huffman table by counting the symbols within several training files. While the Compressor was optimized for speed, decompression was only implemented to ensure the algorithm is working correctly. Hence, Huffman coding will not be part of the energy tests in section 4 and its decompression speeds are omitted in section 3.4. Since we try to provide a static tree for many documents (e.g. for all html files with English content), it makes sense to exchange the required tree once in advance only and use it for several transactions. Therefore our implementation will not include the used tree within its output, saving some additional bytes. Within this document several Huffman trees will be used whereby $huffman_{html}$ and $huffman_{js}$ are trees derived from all html / js payloads of P_{24} (see section 2.2). $Huffman_{gen}$ will be, in general, any Huffman tree derived from any sort of payload using the algorithm described above. All resulting compression ratios are compared with those of an adaptive tree used by gzip in `Z_HUFFMAN_ONLY` mode [Deutsch 1996a].

3.3 LZW-based dictionary

In order to examine dictionary coding we implemented a LZW based algorithm [Welch 1984] (a successor of LZ78 [Ziv and Lempel 1978]) which was inspired by the code from [Nelson 1989]. We used a maximum length of $M=16$ for the dictionary entries and a Trie to retrieve the matches. Our dictionary allows a maximum of 65536 entries whereas every entry will be compressed by using its corresponding 16-bit index. LZ77 based algorithms (like gzip) are expensive on the compression side due to the need to find the longest match from a sliding window [Bell and Kulp 1993]. Using a priori knowledge to create a static dictionary, the missing adaption to changing

payload-content usually degrades the compression ratio. However, compression speed increases as the data-structures needed for efficient retrieval of the longest dictionary match, can be precomputed and do not need (expensive) updating during compression [Bell and Kulp 1993].

As the speed of the current implementation requires further optimization, we only analyzed achievable compression ratios using a static dictionary and omit speed and energy tests. The language dependency of the dictionary is analyzed by using the four front pages of amazon, youtube, wikipedia, and facebook. Those websites were chosen because they are common[1], translated into the tested languages German, English, and Japanese and the front pages for all languages differed only in textual content and not in layout or structure. The dictionaries (D_{de}, D_{en}, D_{jp}) each use the four front pages of their corresponding language as do the payloads (P_{de}, P_{en}, P_{jp}). Since those websites heavily use `<script>` and `<style>` another dictionary $D_{-script}$, based on each language with those tags removed and a dictionary $D_{-content}$ with additionally removed textual content, will be tested as well.

3.4 Results

3.4.1 Chunk size

The impact of different chunk sizes on the general purpose algorithms was measured to find proper values for the following energy tests. We compressed the html part of payload P_{24} (see 2.2) using each of the algorithms. The chunk size is the maximum number of bytes that each compressor received at once per file. Depending on the file size (see Table 3 for average file sizes in P_{24}), this means fewer calls to the compressor API with larger to-be-compressed data chunks. It also leads to less overhead for the compression format because of fewer compressed data chunks. Larger chunk sizes, however, also require more memory on the system and increased memory management efforts for the application. Figure 3 and Figure 4 show the resulting impact on compression ratio and compression speed respectively.

The libraries for gzip and $lzma_1$ provide a stream-like API that performs internal buffering, so their compression ratio is not affected by the chosen chunk size. Compression speed is slightly lower for very small chunks, which seems to be the function call overhead of the API. Decompression is not affected since the compression stream always provides compressed data chunks of the same size, except for end-of-file situations. With the exception of the necessary function call overhead, both Huffman based compressors are unaffected by the chunk's size and offer slightly faster compression with larger chunks only.

The compression ratio of $quicklz_1$ is slightly worse until reaching chunk sizes of about 4 KiB, but stays stable from there on. This seems to be caused by the constant format overhead per chunk. However, the average compression ratio is better than that of e.g. lzo_1, which is probably a result of the optional history-buffer. Unlike the asynchronous stream-like interface of $gzip_6$ and $lzma_1$, $quicklz_1$ returns a compressed data chunk immediately after each call to the API. Compression and decompression speed each double between 256 bytes and 64 KiB chunk sizes. This increase progresses linearly and is too small to be caused by function calls. Instead, the algorithm seems to be faster when allowed to compress or decompress larger data chunks at once.

lzo_1 and lz4 both provide a very simple to use block based API. Every time their API is called to compress a data block the compressed result is returned directly after the function call. They do not keep track of internal state between calls nor use a global context. This design benefits greatly from increased chunk sizes both for compression ratio and speed. It would probably not be difficult to design a stream-like API similar to the one of $gzip_6$ around these algorithms, so that they would work efficiently independent of external chunk sizes. But that is outside the scope of this paper. The compression ratio of the two algorithms is practically identical. Speed-wise however, lz4 shows superior performance to lzo_1, especially for decompression.

[1] http://www.alexa.com/topsites/countries/DE

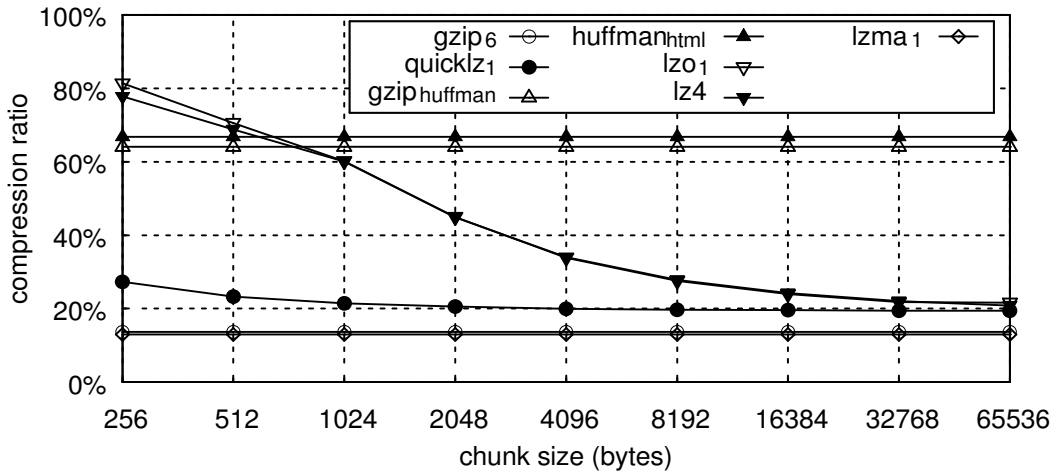

Figure 3: Chunk size impact on html compression ratio using P_{24} (less is better)

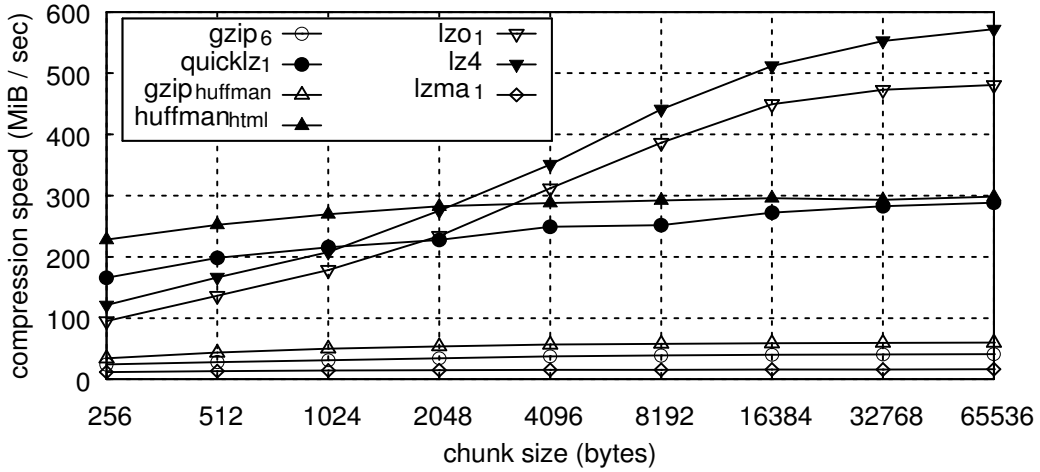

Figure 4: Chunk size impact on html compression speed using P_{24} (more is better)

3.4.2 Content type

Given the results of the previous tests we decided to settle for a chunk size of 64 KiB for the compression tests split by content type. The measurements were performed on the payload P_{24} using machine M_{a1}. The compression tests were performed using an extension of the NetAnalyzer tool (see section 2.1). Although the compression itself is single-threaded, we used `pthread_setaffinity_np` to bind the compression thread to one core. This increased reproducibility and reduced the number of cache misses [Love 2003].

The results of the content type tests in Table 5 to Table 7 complement the measurements shown in Figure 3 and Figure 4. Best results are shown in bold. These tables include stats for the fastest ($gzip_1$) and best ($gzip_9$) gzip compression level. For us, the takeaway from these tests is: $lzma_1$ and $gzip_9$ provide the best compression ratio, which is consistent with what Figure 3 indicated for html-only data. $quicklz_1$, lzo_1, and especially lz4 are superior when focused on speed. As expected, the compression ratio of both static Huffman codings $huffman_{html}$ and $huffman_{js}$ is inferior to that of the adaptive version used in gzip [Crochemore and Lecroq 2010].

	html	xml	css	js	json
$gzip_1$	0.157	0.151	0.242	0.371	0.319
$gzip_6$	0.137	0.131	0.201	0.325	0.282
$gzip_9$	0.136	0.128	**0.191**	0.325	**0.280**
$gzip_{huffman}$	0.640	0.624	0.640	0.652	0.673
$quicklz_1$	0.195	0.186	0.302	0.453	0.403
lzo_1	0.216	0.206	0.319	0.481	0.420
lz4	0.208	0.200	0.337	0.494	0.410
$lzma_1$	**0.130**	**0.121**	0.209	**0.323**	0.294
$huffman_{html}$	0.668	0.692	0.730	0.780	0.772
$huffman_{js}$	0.757	0.790	0.727	0.692	0.782

Table 5: Compression ratio for P_{24} using 64 KiB chunks (less is better)

	html	xml	css	js	json
$gzip_1$	56.8	63.7	37.4	29.8	27.2
$gzip_6$	30.3	34.3	20.1	17.3	17.3
$gzip_9$	24.4	25.9	16.6	12.4	14.5
$gzip_{huff}$	49.0	56.6	46.0	49.3	38.5
$quicklz_1$	216.7	234.9	113.7	110.6	77.9
lzo_1	420.0	411.2	224.3	190.2	195.6
lz4	**505.5**	**507.6**	**270.0**	**220.9**	**229.9**
$lzma_1$	13.7	15.8	8.8	6.7	7.0

Table 6: Compression speed (in MiB/s) for P_{24} using 64 KiB chunks (more is better)

	html	xml	css	js	json
$gzip_1$	230.2	242.4	119.9	118.7	112.0
$gzip_6$	256.5	285.1	143.3	141.9	139.0
$gzip_9$	257.2	281.0	150.2	137.3	144.2
$gzip_{huff}$	116.9	136.8	104.0	110.4	95.1
$quicklz_1$	362.7	441.8	216.3	174.3	173.1
lzo_1	570.4	602.8	291.0	256.4	328.1
lz4	**1120.8**	**1150.3**	**660.5**	**712.8**	**711.5**
$lzma_1$	62.2	71.3	32.4	25.9	26.7

Table 7: Decompression speed (in MiB/s) corresponding to Table 6 (more is better)

Table 5 and Table 6 indicate a correlation between compression ratio and (de)compression speed for some of the algorithms. Therefore we calculated the Bravais-Pearson correlation coefficient between those values using equation 3. The correlation for Huffman based compressors strongly depended on the used payload types, resulted in entirely different values for aforementioned tests, and will be omitted. Table 8 shows that dictionary based compressors have a strong negative correlation between compression ratio and compression speed. Highly compressible files will also consume less time for compression. Moreover, decompression almost showed similar correlation values.

$$[ht] r_{xy} = \frac{\sum_{i=0}^{n}(x_i-\overline{x})(y_i-\overline{y})}{\sqrt{\sum_{i=0}^{n}(x_i-\overline{x})^2 \cdot \sum_{i=0}^{n}(y_i-\overline{y})^2}} \qquad \overline{x} = \frac{1}{n}\sum_{i=0}^{n} x_i \qquad \overline{y} = \frac{1}{n}\sum_{i=0}^{n} y_i \qquad (3)$$

$lzma_1$	lz4	$gzip_9$	$gzip_1$	lzo_1	$gzip_6$	$quicklz_1$
-0.952	-0.944	-0.941	-0.937	-0.924	-0.908	-0.896

Table 8: Bravais-Pearson correlation between compression ratio and compression-speed

In order to reduce the number of possible testing-combinations and to make them comparable with previous results, the two following tests for Huffman and dictionary coding will focus on html payloads only.

3.4.3 Huffman coding

Table 5 concluded that the compression ratio of a static Huffman tree, derived from the complete html traffic of P_{72}, is similar to the ratio of an adaptive gzip. The language dependency of Huffman coding in general is analyzed using the payloads P_{de}, P_{en}, P_{jp} and their corresponding Huffman trees referred to as $huffman_{gen}$. While compression ratios for English and German payloads are almost identical, Japanese payloads seem less compressible using Huffman coding, due to their higher entropies (see Table 9).

	P_{de}	P_{en}	P_{jp}
$huffman_{gen}$	69.0%	68.8%	74.7%
$gzip_{huffman}$	67.3%	67.1%	71.4%

Table 9: Huffman compression ratio per language

	D_{de}	D_{en}	D_{jp}	$D_{-script}$	$D_{-content}$
P_{de}	40.2%	45.3	46.2	48.6% (+8.4%)	51.4% (+2.8%)
P_{en}	44.1%	40.2	45.8	47.7% (+7.5%)	50.4% (+2.7%)
P_{jp}	56.1%	58.5	40.7	49.6% (+8.9%)	56.5% (+6.9%)
P_{24}	48.3%	49.8	52.7	47.9% (-0.4%)	52.2% (+4.3%)

Table 10: LZW-dictionary compression ratio depending on payload

3.4.4 Dictionary

Table 10 shows that P_{24} can be compressed to 48.3% using the dictionary D_{de} which is superior to the 66.7% offered by the Huffman tree. English and German websites can be compressed with another language's dictionary at a small overhead of 5%. However, Japanese websites heavily depend on the language component within the dictionary to achieve good compression ratios. The last two columns respectively use the scriptless / contentless dictionary (see section 3.3) for the language of each row. For P_{24} those two variants are based on the German dictionary because it offered the best compression ratio. Removing scripts and styles from a dictionary of a language results in a compression ratio degraded by about 8%, indicating an extensive usage of `<script>` and `<style>` tags within the examined websites. It should be far more efficient to move those contents to external files, at least when using a LZW-based dictionary. P_{24} seems to prove this hypothesis as the compression ratio using $D_{-script}$ is slightly better (0.4%) than that of D_{de}. However, LZ77 based compressors will, most likely, behave differently as they use a local dictionary instead of a global one. Additionally removing any textual content from each dictionary of a language only keeping the html tags (named $D_{-content}$) increased the compression ratio by another 3% for German / English texts and about 7% for Japanese contents. This indicates that a language-independent static dictionary for html payloads is feasible but slightly less efficient.

3.4.5 General optimizations

Previous tests indicated a huge amount of tabs and spaces within the payloads. Supplementary tests were conducted to analyze whether additional bytes can be saved by replacing indentation tabs with spaces. We examined a fraction of payload P_{24} created by using only those files which solely use tabs for indentation and filtering duplicates referred to as P_{tab}. Table 11 points out those new results. The values state that it doesn't matter whether to use one space or one tab for indentation when working with common compression algorithms. However, using four spaces for indentation is inefficient and increases the traffic by about 4% even when using gzip compression.

	original	tab -> 1 space	tab -> 2 space	tab -> 4 spaces
none	50,626	50,626 (0.0%)	55,026 (+8.7%)	63,826 (+26.1%)
$huffman_{static}$	33,775	32,958 (-2.4%)	34,494 (+2.1%)	36,694 (+8.6%)
$gzip_{huffman}$	32,996	32,370 (-1.9%)	33,626 (+1.9%)	35,363 (+7.2%)
$gzip_1$	10,070	10,065 (0.0%)	10,191 (+1.2%)	10,464 (+3.9%)

Table 11: Impact of tab/space indentation on the compression (in bytes) of payload P_{tab}

Inspired by the growing output size when using multiple spaces for indentation, we also applied a simple minification to the previously tested payload P_{tab} by removing all \r, \n, \t and multiple consecutive whitespaces (see Table 12). However, our simple implementation will, most likely, break javascript code, style sheets or `<pre>` blocks. Moreover, many additional operations as comment-removal or variable-renaming could be applied to further enhance the minification. Widely used CMS and templates pose another challenge for minification. Using those, it is not easily possible to minify a website beforehand as it is composed of numerous elements. Minification within the compressor would be a far better solution here, probably at the cost of lowered compression speeds.

	uncompressed	$huffman_{static}$	$gzip_{huffman}$	$gzip_1$
original	50,626	33,757	32,996	10,070
minified	45,317 (-10.5%)	30,214 (-10.5%)	29,744 (-9.9%)	9,673 (-3.9%)

Table 12: Impact of minification on the compression (in bytes) of payload P_{tab}

3.5 Conclusion

The $gzip_6$ algorithm offers a good trade-off between compression ratio and speed. $lzma_1$ showed strong compression ratio but was also the slowest option throughout the measurements with the highest memory consumption. For some content types $gzip_9$ even provided slightly better compression ratio than $lzma_1$. For these reasons LZMA level 1 and higher might not be suitable for low-latency real-time data compression or decompression. lzo_1 and lz4 are both fast and show similar characteristics (need large chunk size) while lz4 offers superior speed. Given an appropriate chunk size (32 KiB and above), lz4 would also be preferred over $quicklz_1$ as it is faster and provides competitive compression ratio. In a scenario where a large chunk size is not an option and good compression is more important than speed, $quicklz_1$ might be a viable alternative.

Static compression algorithms, using a priori knowledge, offer competitive compression ratio and have room for additional optimization as they rarely need data-structure adjustments during compression. Those algorithms may use data structures otherwise too costly (like Patricia-Tries [Morrison 1968]) to reduce memory consumption rendering them practical for embedded systems where memory and CPU time is expensive. Albeit there are many adjustable variables which need further investigation in order to provide a set of rules for some content-type which maximizes the compression ratio. It is possible to save additional bytes by using minification which could be considered a lossy compression algorithm. Minification is applicable to all of the analyzed compressible content-types (html, js, css, xml, json) and can be quite complex, e.g. when renaming variables to shorter sequences. If suitable, pre-minification should be applied at the cost of edit ability.

4 ENERGY TESTS

This section will pick a subset of the previously evaluated algorithms and measure energy consumption relative to one another using real life the traffic we analyzed before. The goal of these tests is to show how much the energy consumption of different compression algorithms vary. We tried to eliminate many disruptive factors to measure the algorithms' energy consumption in terms of their CPU and network utilization.

4.1 Hardware

The setup of our energy tests consists of three separate hardware devices: the server, the client, and a switch linking them together. The client and server are machines originally intended for regular desktop use; see Table 13 for their specification. They are linked using a regular 100 MBit switch which will be of no concern to the test.

Machine	CPU	RAM	HDD
M_{ec}	Intel Atom N2800	2 GB DDR3	Seagate Momentus 5400.6 250 GB
M_{es}	Intel Core 2 Duo E6550	2 GB DDR2 800	Seagate Momentus 5400.6 250 GB

Table 13: Computers used for energy tests. M_{ec} works as the client, M_{es} works as the server

We tried to eliminate possibly interfering factors to achieve a uniform consumption measurement for idle mode. Unnecessary components like on-board audio, LPT and RS232 ports were disabled in the BIOS. Also, `cpufreq` configured for `ondemand` was used to reduce the

machines' (idle) power consumption. Both test machines are using Gentoo Linux running kernel version 3.8 optimized for each machine's CPU.

We used two Voltcraft Energy Logger 4000 devices to measure the consumed energy. They are connected between the mains socket and the device to be measured. While the energy logger's display provides a precision of 0.1 watts, the exported binary data leads to a lower precision of about 0.23 watts when using an average voltage of 230 volts (230.0V 0.001A = 0.23W).

4.2 Software

The energy tests were run by a tool we wrote for this purpose. NetAnalyzer served well for analysis but for energy tests we needed a program with as little overhead as possible. It can either be run in client or server mode and was deployed on both test machines M_{ec} and M_{es}.

4.2.1 Client mode configuration

When the tool is run in client mode the user can set various parameters to control the server's behavior. We used P_{24} (see section 2.2) for our energy tests to ensure real-world traffic. P_{24} does not only consist of the original payload but also includes meta information about the time stamp of each request. The client parses those meta information to be able to decide which file to request at a time. The tool allows for the payload to be sent in a shorter interval using a replay speed option which makes the test process less time consuming while preserving the original request distribution. P_{24} was used for the energy tests, so the original request time frame was 24 hours. We chose a replay speed of 3x to make each of the tests finish after 8 hours instead of 24. Section 2.2 has shown that the production server we analyzed was not running under full load. In fact P_{24} consists of only about 2.7 GiB of compressible data over a time period of 24 hours. This was not enough traffic for our measurements. We decided to simulate more load by sending each request multiple times, so each file was repeatedly requested 26 times in a row. The repeated requests of one file were equally spread over the time between two real requests. A chunk size option tells the server the size of chunks to use when reading a file and pass it to the compression algorithms (see 3.4). The compressed chunk is then sent to the client. We decided to run the energy tests using the three most diverse algorithms evaluated in section 3: $gzip_6$, $lzma_1$, and lz4. In addition tests were also run without compression for reference, referred to as none.

4.2.2 Server mode configuration

The server mode is initialized with a link to the actual payload P_{24} on the local file system. P_{24} consists of payload that the NetAnalyzer exported, where each HTTP response's payload was exported as a single file. Every time a client connects and requests a file, the server reads the file from the file system (xfs) using the given chunk size, compresses it with the appropriate compression algorithm and sends the data to the client one compressed chunk after another. Each access to the local file system made by the server (reading chunks) used the direct IO flag `O_DIRECT`. This forces the operation system to always access the drive directly without using the file system cache even when the same file is read repeatedly. While this is unusually in production environments, it ensures better reproducibility.

4.2.3 Protocol overhead

The energy tests contain a protocol overhead on top the actual payload. The client's overhead consists of the client telling the server which file to send, chunk size and compression algorithm. The server on the other hand sends the client the length of the following compressed chunk. This overhead, however, is negligible as it makes up only 0.25% of the whole communication for client and server combined for $gzip_6$ compression with a chunk size of 64 KiB. Table 14 presents the most important energy test configuration options that were explained in this section.

Test Runtime	8 hours
Repeat count	26x
Chunk size	64 KiB
Compression	none, $gzip_6$, $lzma_1$, lz4
Data	P_{24} (2.7 GiB, 75,278 files)
Content types	html, js, css (see Table 3 for distribution details)

Table 14: Energy test parameters and statistics

4.3 Results

Table 15 (left) shows the power consumption measurements collected for the client M_{ec}. The tests were run using the configuration described in Table 14: 69 GiB of uncompressed payload spread over 1.9 Mio separate connections over a time frame of 8 hours. The actual amount of bytes sent over the network was less when using compression, but the number of connections stayed the same. The client received the data being sent from the server and decompressed it. The difference in energy consumption between compression algorithms was barely measurable with our equipment. Most of the values are located in an area only about 0.3 watts wide, which is just slightly higher than the measurement accuracy of our energy logger (see section 4.1).

The values presented in Table 15 (left) show noticeable trends. The idle consumption is in fact the lowest, although not by much, and $lzma_1$ decompression measurements are the highest. Although Table 16 shows that the client spent about 100min (6010s) for decompressing data using $lzma_1$. Table 15 indicates almost no increase in energy consumption compared to lz4 decompression, which kept decompression only about 7min (426s) over a time frame of 8 hours.

The server measurements presented in Figure 5 are more diverse, since the server is responsible for the compression workload. The idle measurement is almost a straight line within the energy logger's accuracy. The traffic distribution of P_{24} can be recognized by looking at the graph, especially for $lzma_1$, which consumes considerably more energy than the other algorithms. $gzip_6$ replicates the payload traffic although by a smaller degree than $lzma_1$. The values for none and lz4 both show nearly identical progression but are lower than $gzip_6$ and $lzma_1$ at any time. These impressions are confirmed by the energy consumption details in Table 15 (right). Compressing payload with lz4 does not increase the energy consumption of our server, the data rather shows it might even be less costly (44.90 watts vs. 44.80 watts). This might be an effect of the reduced network traffic caused by compression, which leads to the theory that energy-wise the traffic savings from lz4 compression are higher than costs of the compression itself.

In addition to the measurements made by the energy logger, we analyzed the CPU usage of the client and server machines during the energy tests. The rationale behind this was to verify that power consumption and CPU utilization were consistent. The CPU usage was recorded by continuously reading the output of `/proc/stat` on each system. Figure 6 and Figure 7 show the CPU usage over time on the client. Table 17 (left) lists more details on the CPU usage with user and irq (software and hardware interrupts combined) values separated. For the server, Figure 8, Figure 9, and Table 17 (right) present the same information respectively.

Overall, the CPU usage and energy measurements are not only consistent with each other, we also noticed good reproducibility during the test.

	kWh	voltage avg (σ)	watt avg (σ)
idle	0.095	222.6 (1.45)	11.99 (0.092)
none	0.096	222.9 (0.90)	12.09 (0.107)
$gzip_6$	0.096	222.8 (1.05)	12.05 (0.112)
$lzma_1$	0.096	222.6 (1.36)	12.14 (0.105)
lz4	0.096	221.9 (1.35)	12.08 (0.102)

kWh	voltage avg (σ)	watt avg (σ)
0.348	223.4 (1.44)	43.82 (0.200)
0.357	223.7 (0.90)	44.90 (0.362)
0.365	223.6 (1.05)	45.92 (0.734)
0.379	223.4 (1.37)	47.62 (1.313)
0.356	222.7 (1.35)	44.80 (0.383)

Table 15: Energy consumption details using options described in Table 14 on M_{ec} (left) and M_{es} (right)

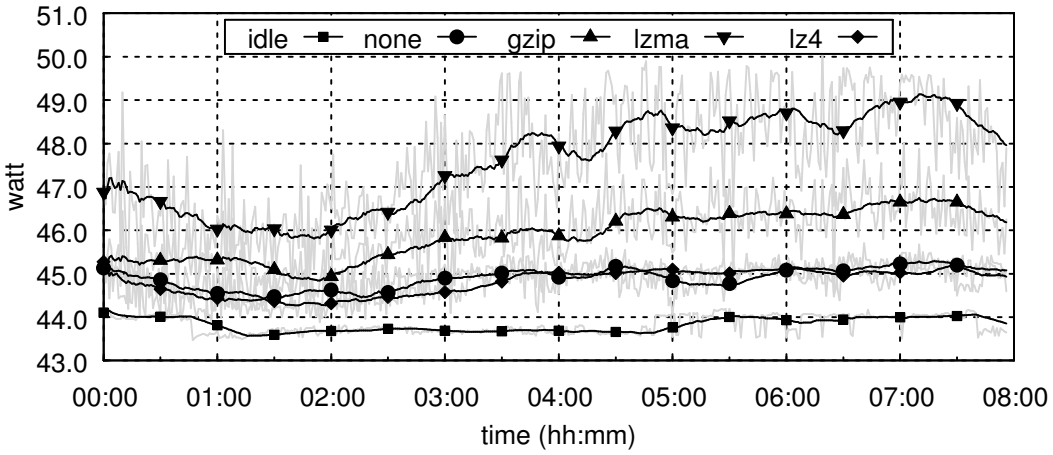

Figure 5: Power consumption using options described in Table 14 on M_{es} (30min average)

4.4 Conclusion

The measurements on our client machine M_{ec} did not provide significant results. The reason might be decompression being so cheap in general (see Table 7) or because our client machine was already very energy efficient. Probably both factors were responsible, but we were not able to conclusively determine the exact degree of each.

	uncompressed (MiB/s)	compressed (MiB/S)	compr. time (s)	decompr. time (s)
none	2.467	-	-	-
$gzip_6$	2.467	0.440	2933	1451
$lzma_1$	2.467	0.427	5985	6010
lz4	2.467	0.673	235	426

Table 16: Network traffic using options from Table 14 for compression on M_{es} (left) and decompression on M_{ec} (right)

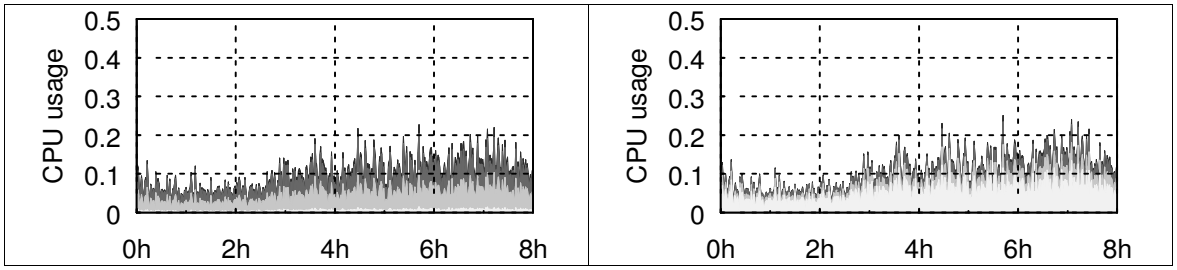

Figure 6: CPU usage over time using Table 14 on M_{ec} with none (left) and $gzip_6$ (right)

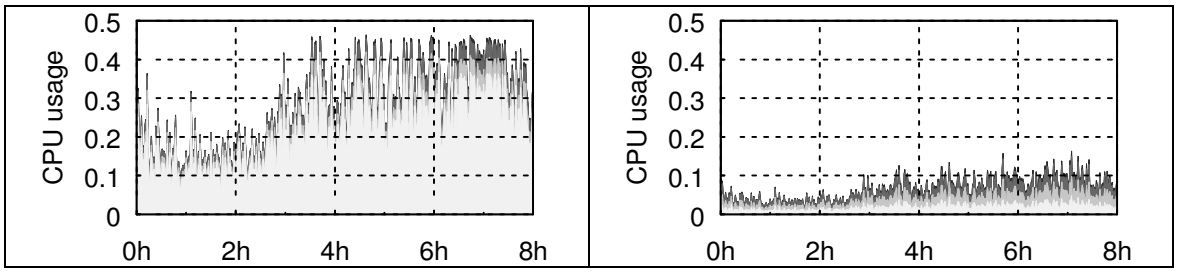

Figure 7: CPU usage over time using Table 14 on M_{ec} for $lzma_1$ (left) and lz4 (right)

The measurements on the server machine M_{es} show that the power consumption development of $gzip_6$ and $lzma_1$ is clearly dependent on the traffic distribution. Although being relatively energy inefficient, both offer good compression ratio. LZ4, however, did not lead to an increased energy consumption or CPU usage compared to tests without compression. LZ4 lead to higher user space utilization during compression but the system spent less time handling interrupts (see user and irq column in Table 17). Compared to tests without compression, our theory is that the increased power consumption for lz4 compression was balanced by the saved network traffic.

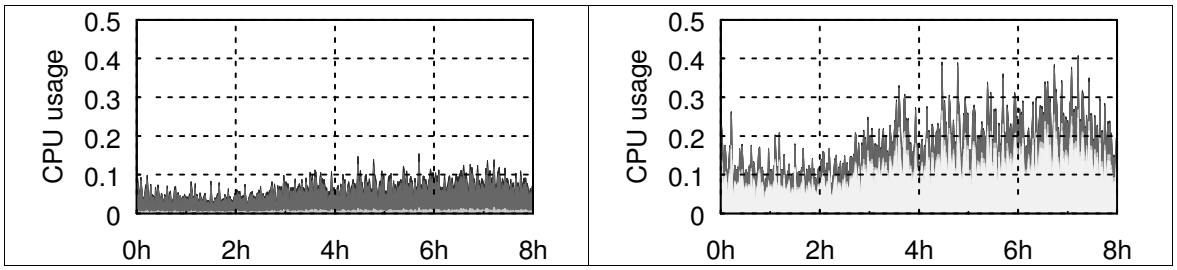

Figure 8: CPU usage over time using Table 14 on M_{es} for none (left) and $gzip_6$ (right)

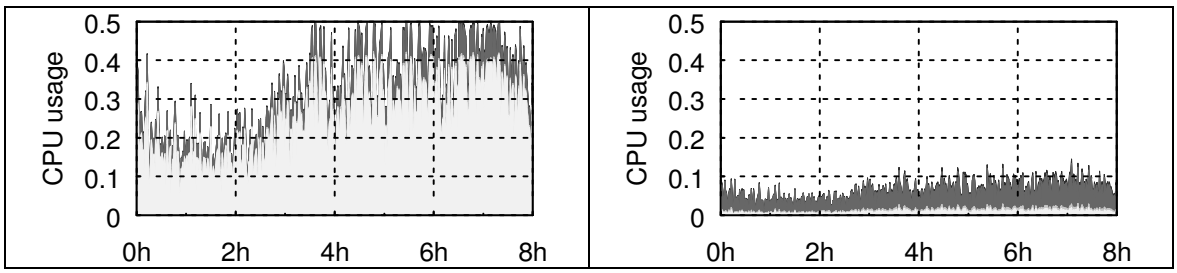

Figure 9: CPU usage over time using Table 14 on M_{es} for $lzma_1$ (left) and lz4. (right)

	all (%)	user (%)	irq (%)
none	5.437	0.581	0.885
$gzip_6$	8.000	5.563	0.475
$lzma_1$	24.423	21.360	0.526
lz4	6.127	2.364	0.505

all (%)	user (%)	irq (%)	iowait (%)
5.944	0.148	0.876	4.490
16.223	10.360	0.394	4.554
29.826	24.452	0.419	4.593
5.955	0.907	0.426	4.250

Table 17: CPU usage statistics using options described in Table 14 on M_{ec} (left) and M_{es} (right)

5 FUTURE WORK

Although we examined the feasibility of static compression methods we did not provide a final Huffman tree / dictionary usable e.g. for all German websites. This could be a subject for supplementary analysis to come up with a suitable solution.

Our energy tests concluded that the energy consumption of LZ4 is similar to using no compression. More sophisticated measuring equipment could conclude whether it is possible for compression to even reduce energy consumption. Another interesting test bed could use wireless instead of wired connections. The relation between energy savings due to traffic reduction and the additional energy needed for compression should be more evident than in our test setup.

We examined the complexity needed for data compression but omitted the CPU overhead spent by the underlying scripting language to generate websites. Further tests should be conducted to examine the overhead needed for compression using the algorithms directly on the target systems including dynamic website creation. Those could point out the relation between the

amount of time needed for website creation and compression. This would also require adjusting web servers and browsers to support the to-be-tested algorithms. Arvind Jain and Jason Glasgow looked into part of this problem, focusing on why the benefits of content compression often do not lead to better load times for web pages [Jain and Glasgow 2012].

REFERENCES

BELL, T. C. AND KULP, D. 1993. Longest-match string searching for ziv-lempel compression. *Softw., Pract. Exper. 23*, 7, 757–771.

BELSHE, M., PEON, R., THOMSON, M., AND MELNIKOV, A. 2013. Hypertext transfer protocol version 2.0. RFC (Draft).

BLELLOCH, G. E. Introduction to data compression. Tech. rep., Carnegie Mellon University.

BUTLER, J., LEE, W.-H., MCQUADE, B., AND MIXTER, K. 2008. A Proposal for Shared Dictionary Compression over HTTP. Tech. rep.

CROCHEMORE, M. AND LECROQ, T. 2010. Algorithms and theory of computation handbook. Chapman & Hall/CRC, Chapter Text data compression algorithms, 14–14.

DEUTSCH, P. 1996A. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational).

DEUTSCH, P. 1996B. GZIP file format specification version 4.3. RFC 1952 (Informational).

DEUTSCH, P. 1996C. ZLIB Compressed Data Format Specification version 3.3. RFC 1950 (Informational).

FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. 1999. Hyper-text transfer protocol – http/1.1. RFC 2616 (Standard).

HUFFMAN, D. A. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers* 40, 9 (September), 1098–1101.

JAIN, A. AND GLASGOW, J. 2012. Use compression to make the web faster.

LOVE, R. 2003. Kernel korner: CPU affinity. *Linux J. 2003*, 111 (July), 8–.

MORRISON, D. R. 1968. Patricia – practical algorithm to retrieve information coded in alphanumeric. *J. ACM 15*, 4 (Oct.), 514–534.

NELSON, M. R. 1989. LZW data compression. *Dr. Dobb's J. 14*, 10 (Oct.), 29–36.

PAPADOGIANNAKIS, A., VASILIADIS, G., ANTONIADES, D., POLYCHRONAKIS, M., AND MARKATOS, E. P. 2012. Improving the performance of passive network monitoring applications with memory locality enhancements. *Comput. Commun. 35*, 1 (Jan.), 129–140.

PEON, R. AND RUELLAN, H. 2013. Http/2.0 header compression. RFC (Draft).

POSTEL, J. 1981. Transmission control protocol. RFC 793 (Standard).

POSTEL, J. 1983. The tcp maximum segment size and related topics. RFC 879 (Unknown).

WELCH, T. A. 1984. A technique for high-performance data compression. *Computer 17*, 6 (June), 8–19.

YANG, L., DICK, R. P., LEKATSAS, H., AND CHAKRADHAR, S. 2010. High-performance operating system controlled online memory compression. *ACM Trans. Embed. Comput. Syst. 9*, 4 (Apr.), 30:1–30:28.

ZIV, J. AND LEMPEL, A. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3, 337–343.

ZIV, J. AND LEMPEL, A. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory 24*, 5, 530–536.