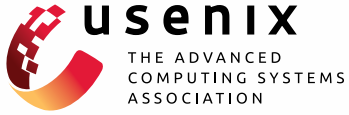


Automatic and Dynamic Configuration of Data Compression for Web Servers

Eyal Zohar, ***Yahoo! Labs;*** **Yuval Cassuto,** ***Technion—Israel Institute of Technology***

https://www.usenix.org/conference/lisa14/conference-program/presentation/zohar

This paper is included in the Proceedings of the 28th Large Installation System Administration Conference (LISA14).

November 9–14, 2014 • Seattle, WA

ISBN 978-1-931971-17-1

Automatic and Dynamic Configuration of Data Compression for Web Servers

Eyal Zohar*
Yahoo! Labs
Haifa, Israel
eyalz@yahoo-inc.com

Yuval Cassuto
Technion - Israel Institute of Technology
Department of Electrical Engineering
Haifa, Israel
ycassuto@ee.technion.ac.il

Abstract

HTTP compression is an essential tool for web speed up and network cost reduction. Not surprisingly, it is used by over 95% of top websites, saving about 75% of web-page traffic.

The currently used compression format and tools were designed over 15 years ago, with static content in mind. Although the web has significantly evolved since and became highly dynamic, the compression solutions have not evolved accordingly. In the current most popular web-servers, compression effort is set as a global and static compression-level parameter. This parameter says little about the actual impact of compression on the resulting performance. Furthermore, the parameter does not take into account important dynamic factors at the server. As a result, web operators often have to blindly choose a compression level and hope for the best.

In this paper we present a novel elastic compression framework that automatically sets the compression level to reach a desired working point considering the instantaneous load on the web server and the content properties. We deploy a fully-working implementation of dynamic compression in a web server, and demonstrate its benefits with experiments showing improved performance and service capacity in a variety of scenarios. Additional insights on web compression are provided by a study of the top 500 websites with respect to their compression properties and current practices.

1 Introduction

Controlling the performance of a web service is a challenging feat. Site load changes frequently, influenced by variations of both access volumes and user behaviors. Specifically for compression, the load on the server also depends on the properties of the user-generated content (network utilization and compression effort strongly depends on how compressible the data is). To maintain good quality-of-experience, system administrators must monitor their services and adapt their configuration to changing conditions on a regular basis. As web-related technologies get complex, ensuring a healthy and robust web service requires significant expertise and constant attention.

The increasing complexity raises the popularity of automated solutions for web configuration management [7, 25]. Ideally, an automatic configuration management should let system administrators specify high-level desired behaviors, which will then be fulfilled by the system [32]. In this paper we add such automation functionality for an important server module: HTTP compression.

HTTP compression is a tool for a web-server to compress content before sending it to the client, thereby reducing the amount of data sent over the network. In addition to typical savings of 60%-85% in bandwidth cost, HTTP compression also improves the end-user experience by reducing the page-load latency [29]. For these reasons, HTTP compression is considered an essential tool in today's web [34, 5], supported by all web-servers and browsers, and used by over 95% of the leading websites.

HTTP compression was standardized over 15 years ago [9], and with static web pages in mind, i.e., suitable for "compress-once, distribute-many" situations. But the dynamic nature of Web 2.0 requires web-servers to compress various pages on-the-fly for each client request. Therefore, today's bandwidth benefits of HTTP compression come with a significant processing burden.

The current most popular web-servers [28] (Apache, nginx, IIS) have an easily-deployable support for compression. Due to the significant CPU consumption of compression, these servers provide a configurable compression effort parameter, which is set as a global and static value. The problem with this configurable parameter, besides its inflexibility, is that it says little about

*Also with Technion - Israel Institute of Technology.

the actual amount of CPU cycles required to compress the outstanding content requests. Furthermore, the parameter does not take into account important factors like the current load on the server, response size, and content compressibility. As a result, web operators often have to blindly choose a compression level and hope for the best, tending to choose a low-effort compression-level to avoid overloading or long latencies.

Given its importance, HTTP compression has motivated a number of prior studies, such as [30, 6, 15]. However, our work is unique in considering simultaneously all aspects of compressed-web delivery: CPU, network bandwidth, content properties and server architectures. This combined study, and the concrete software modules provided along with it, are crucial to address the complexity of today's web services [18].

In this paper we present a deployable elastic compression framework. This framework solves the site operator's compression dilemma by providing the following features: 1) setting the compression effort automatically, 2) adjusting the compression effort to meet the desired goals, such as compression latency and CPU consumption, and 3) responding to changing conditions and availability of resources in seconds. We emphasize that the thrust of this framework is not improved compression algorithms, but rather a new algorithmic wrapping layer for optimizing the utilization of existing compression algorithms.

For better understanding of the problem at hand, Section 2 briefly surveys the key challenges of dynamic HTTP compression in cloud platforms. Since compression performance strongly depends on the content data itself, in Section 3 we analyze HTTP content from the top global websites, illuminating their properties with respect to their compression size savings and required computational effort. Then we turn to describe our solution. First we present in Section 4 a fully-functional implementation along with its constituent algorithms. Then, using a real-life workload, in Section 5 we demonstrate how the implementation operates. Section 6 reviews background of HTTP compression and related work, and Section 7 concludes and discusses future work.

2 Challenges

This section sketches some of the challenges of data compression in the dynamic content era. These challenges set the ground for the study that follows in subsequent sections.

2.1 Static vs. Dynamic Compression

Static content relates to files that can be served directly from disk (images/videos/CSS/scripts etc.). Static compression pre-compresses such static files and saves the compressed forms on disk. When the static content is requested by a decompression-enabled client (almost every browser), the web server delivers the pre-compressed content without needing to compress the content upon the client's request [27]. This mechanism enables fast and cheap serving of content that changes infrequently.

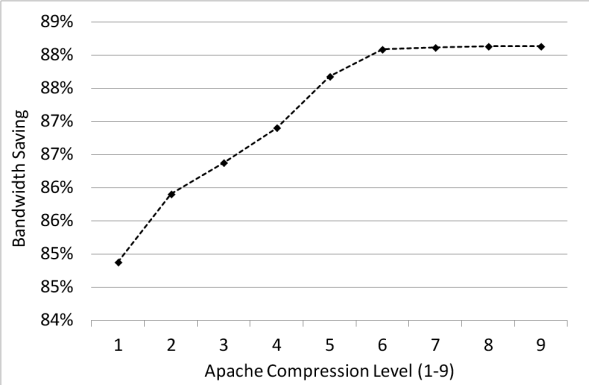
Dynamic content, in the context of this paper, relates to web pages that are a product of application frameworks, such as ASP.NET, PHP, JSP, etc. Dynamic web pages are the heart of the modern web [31]. Since dynamic pages can be different for each request, servers compress them in real time. As each response must be compressed on the fly, the dynamic compression is far more CPU intensive than static compression. Therefore, when a server is CPU bound it may be better not to compress dynamically and/or to lower the compression effort. On the other hand, at times when the application is bound by network or database capabilities, it may be a good idea to compress as much as possible.

2.2 CPU vs. Bandwidth Tradeoff

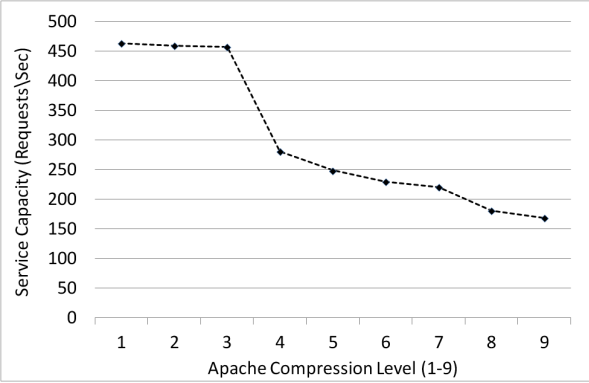
The focus in this paper is on compression of dynamic content, namely unique HTML objects generated upon client request. The uniqueness of these objects may be the result of one or more of several causes, such as personalization, localization, randomization and others. On the one hand, HTML compression is very rewarding in terms of bandwidth saving, typically reducing traffic by 60-85%. On the other hand, each response needs to be compressed on-the-fly before it is sent, consuming significant CPU time and memory resources on the server side.

Most server-side solutions allow choosing between several compression algorithms and/or effort levels. Generally speaking, algorithms and levels that compress better also run slower and consume more resources. For example, the popular Apache web-server offers 9 compression setups with generally increasing effort levels and decreasing output sizes.

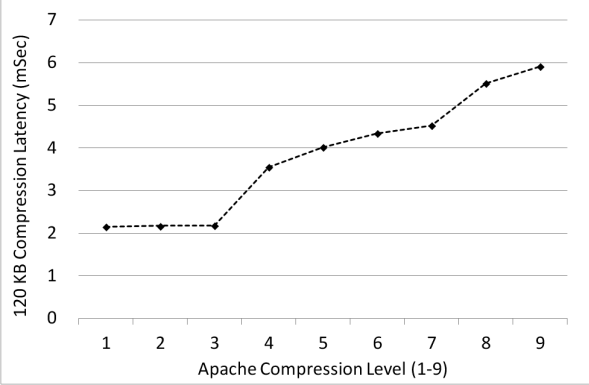
Figure 1a presents a typical bandwidth reduction achieved with all the 9 compression levels in an Apache site. We intentionally postpone the full setup details to Section 5, and just mention at this point that the average page size in this example is 120 KB. The complementary Figure 1b shows the CPU vs. bandwidth trade-off; the higher compression efforts of the upper levels immediately translate to lower capacities of client requests.



(a) Gain - bandwidth saving improves in upper levels



(b) Pain - service capacity shrinks in upper levels



(c) Pain - compression latency grows in upper levels

Figure 1: Compression gain and pain per level, of dynamic pages with average size of 120 KB

2.3 Latency: First-Byte vs. Download-Time Tradeoff

Web testing tools often use the Time To First Byte (TTFB) measurement as an indication of the web server's efficiency and current load. The TTFB is the time from when a browser sends an HTTP request until it gets the first byte of the HTTP response. Some tools [12] practically grade the web-server "quality" according to the TTFB value.

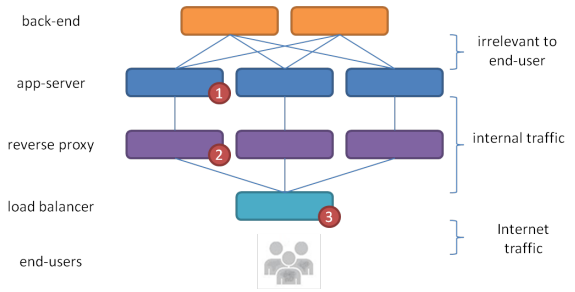


Figure 2: Compression location alternatives in the web server side.

When compression is in use by the server, the TTFB tends to get higher. This is because today's dynamic servers usually perform the following steps in a pure sequential manner: 1) page generation, 2) compression, and 3) transfer. Therefore, the larger the page and the higher the compression level, the larger the TTFB.

On the other hand, compression obviously reduces dramatically the complete download time of a page. Altogether, although the compression increases the TTFB, there is no doubt that this extra delay pays itself when considering the complete download time [14]. The open question is how high the compression level should be to reap download-time benefits without sacrificing latency performance. For example, Figure 1c shows the compression time of a 120 KB page, where the slowest level takes x3 more time than the fastest level, which is a typical scenario as we show in the sequel.

2.4 Where to Compress

Optimizing data compression in the web server is very promising, but simultaneously challenging due to the great richness and flexibility of web architectures. Even the basic question of *where* in the system compression should be performed does not have a universal answer fitting all scenarios. Compression can be performed in one of several different layers in the web server side. Figure 2 illustrates a typical architecture of a web-application server, where each layer may be a candidate to perform compression: 1) the application-server itself, 2) offloaded to a reverse-proxy, or 3) offloaded to a central load-balancer. On first glance all these options seem equivalent in performance and cost implications. However, additional considerations must be taken into account, such as a potential difficulty to replicate application-servers due to software licensing costs, and the risk of running CPU-intensive tasks on central entities like the load-balancer.

3 Web-Compression Study

In this work our focus is on compression of dynamic HTTP content in web-server environments. A study of compression has little value without examining and reasoning about the data incident upon the system. Therefore, in this section we detail a study we conducted on real-world HTTP content delivered by servers of popular web-sites. The results and conclusions of this study have shaped the design of our implementation and algorithms, and more importantly, they motivate and guide future work on algorithmic enhancements that can further improve performance. Another contribution of this study is the good view it provides on current compression practices, which reveals significant inefficiencies that can be solved by smarter compression.

The study examines HTML pages downloaded from top 500 global sites. The content of the pages is analyzed in many aspects related to their compression effort and size savings.

Table 1: Websites analysis - basic properties

Sites supporting "gzip" format	92%
Sites supporting "deflate" format	9%
Average download size (compressed only)	22,526
Average uncompressed file	101,786
Compression ratio (best-median-worst)	11%-25%-53%
Fully dynamic sites	66%

Table 2: Websites analysis - web-server survey.

Developer	Share
Apache	40.3%
nginx	23.9%
gws	15.3%
Microsoft	6.5%
lighttpd	1.7%
YTS	0.9%
PWS	1.1%
Others	10.2%

3.1 Setup

We fetched the list of the top 500 global sites from Alexa [2] in October 2012. For each site, we downloaded its main page at least once every hour with a gzip-enabled browser and saved it for further processing. Then, we emulated possible compression operations performed by the origin servers, by compressing the decompressed form of the pages using various tools and parameters. The analysis presented here is based on 190 consecutive downloads from each site in a span of one week.

3.2 Content Analysis

The first layer of the study is understanding the properties of popular content. Here we are interested in their compression ratios, their dynamism, and statistics on how website operators choose to compress them. We ran a basic analysis of 465 sites out of the top 500 sites. The rest are sites that return content that is too small to compress. A summary is presented in Table 1, where the numbers aggregate the entire set of one week snapshots from all the sites.

We start with examining what the supported compression formats are, by trying each of the formats maintained by IANA[21]. We find that the vast majority of the sites (85%) support "gzip" only, while 9% of that group support "gzip" and "deflate" only. (In the current context "gzip" and "deflate" in quotation marks refer to standard format names. The same terms are also used in other contexts to describe compression implementations, as explained in Section 6.) This means that our choice of gzip and deflate for our implementation and experiments is applicable to the vast majority of real-world HTTP content. As far as compression-ratio statistics go, the median compression ratio (across sites) is 25% (4:1 compression). The best compression ratio we measured is 11% (9:1) and the worst is 53% ($\sim$2:1).

The next measurement of interest is the dynamism of web content, which is the variations of supplied data in time and across requesting clients. We found that 66% of the sites generated a unique page every time we have downloaded a snapshot. This is not surprising considering that dynamic pages are at the heart of Web 2.0. While this sample does not give an accurate prediction, it does attest to the general need for on-the-fly compression pursued in this paper.

An important statistic, especially for the choice of an implementation environment for our algorithms, relates to the server types used by popular sites. The most popular web-server turns out to be Apache, as illustrated in Table 2. These findings match an elaborate survey conducted in November 2012 [28]. This finding was the motivator to choose Apache as the platform for implementation and evaluation of our algorithms.

3.3 Performance Analysis

Beyond the analysis of the compression ratios and existing compression practices, it is useful for our framework to study how compression performance depends on the actual content. The results of this study hold the potential to guide server operators toward adopting compression practices with good balancing of effort and size savings. For that study we used our utilities to run performance

Table 3: Websites performance analysis.

Sites that probably use zlib	51%
Average zlib compression level	5.02
Average added traffic if used fastest level	+9.93%
Average reduced traffic if used best level	-4.51%
Sites using gzip's default (average 5.9-6.1)	200 (47%)

tests on the pages we had downloaded. These tools are available too for download from the project's site [36].

A summary is presented in Table 3, and later expanded in subsequent sub-sections. From the table it is learned that the zlib library emerges as a popular compression library in use today. We reached this conclusion by generating many different compressed forms of each downloaded page using different applications, and comparing the locally generated forms to the one we had downloaded. When a match is found, we project that the server is using the matching compression code and the parameters we use locally. Indeed, these tests show that at least 51% of the sites use zlib somewhere in their system: in the server, reverse-proxy, load-balancer, or at an external offload box. That is another reason, in addition to the popularity of Apache, for using Apache and zlib for the web-compression analysis below.

3.3.1 Compression level

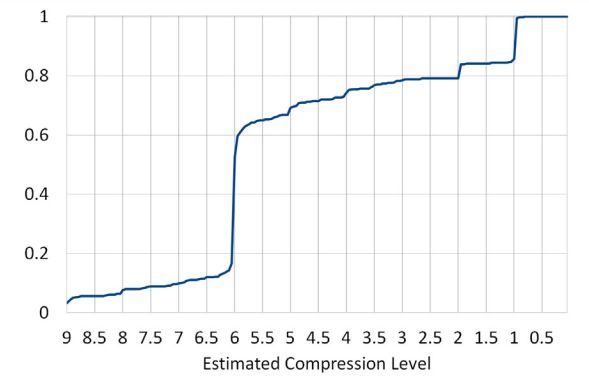
In this sub-section we estimate which compression level is used by each site. The results are presented in Figure 3a as a CDF curve. We found that the average estimated compression level in use is 5.02 and the median is 6, which is also zlib's default level.

3.3.2 Compression effort

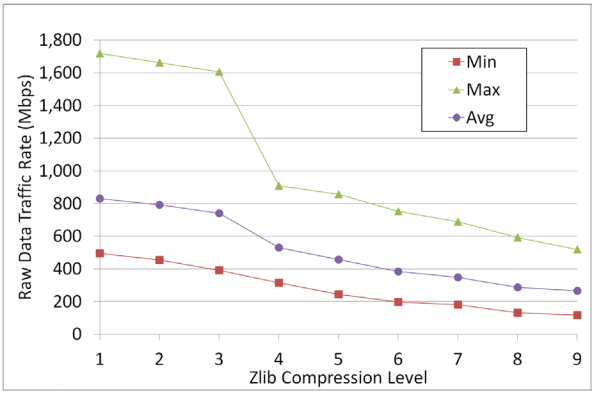
We compressed the contents from all the sites using all 9 levels, and examined the CPU effort induced by the compression levels. The results in Figure 3b show significant effort variation across sites per level. For example, in level 1, the slowest site (the one that required the maximal effort) required x3.5 more CPU power than the fastest site at the same level. In addition, there is at least one case in which level 1 in one site runs slower than level 9 for another site. Hence we conclude that the algorithmic effort, exposed to the user in the form of compression levels, cannot be used as a prediction for the CPU effort.

3.3.3 Fastest vs. slowest levels

Levels 1 and 9 are the extreme end points of the compression capabilities offered by zlib. As such, it is interesting to study the full tradeoff window they span between the



(a) Compression levels used in practice, assuming that the sites use zlib-based compression code



(b) CPU effort induced by each compression level (min, average, and max)

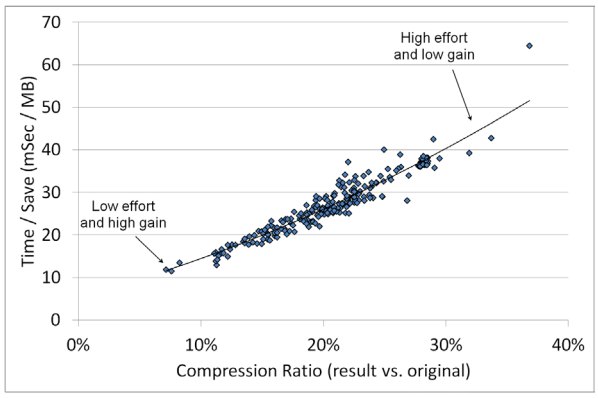
Figure 3: Top sites analysis - levels and CPU effort.

fastest compression and the best-ratio one. Specifically, how much more effort is needed to move from level 1 to level 9, and what the gain is. The answer to this question is presented for all the sites in Figure 4a. The results show, again, that effort cannot be predicted based upon content size and compression level alone: effort can grow by x2 to x6.5 (y locations of the points), while the gain in traffic reduction is anything between 9% to 27% (x locations of the points).

3.3.4 Compression speed vs. ratio

Another important finding for our framework is that the amount of redundancy in the page is highly correlated with low-effort compression, even at the upper levels. When the redundancy is high, zlib is able to find and eliminate it with little effort, thus reducing file sizes at low processing costs. These relations are depicted in Figure 4b, which presents the compression speed (high speed = low effort) versus the compression ratio (low ratio = strong compression) of all the sites we examined, when compressed locally with the default level 6.

(a) Comparing the costs and gains in moving from level 1 to level 9



(b) Time/save ratio vs. page compressibility, if all sites were using zlib level 6 (default)

Figure 4: Top sites analysis - costs, gains, and compressibility.

3.4 Considering the Cloud Pricing Plan

Continuing the study in the direction of web service over the cloud, we are now interested to find for each compression level the totality of the operational costs when deployed over a cloud service, i.e., the *combined* cost of computing and network bandwidth. For this part of the study we assumed an Amazon EC2 deployment, with the prices that were available at the time of the experiment. Clearly the results depend on the instantaneous pricing, and as such may vary considerably. Thus the results and conclusions of this study should be taken as an illustrative example. We further assume that the offered cloud services are fully scalable, stretching to unlimited demand with a linear increase in operational costs. Figure 5 shows the optimal level on a per-site basis, revealing that level 7 is the optimal level for most sites, but also showing that some sites gain more from level 6 or 8.

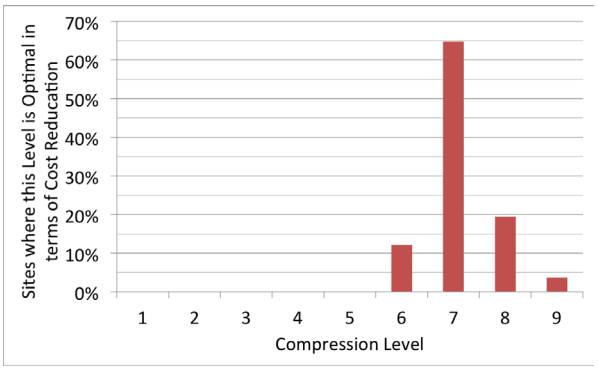


Figure 5: Top sites analysis - considering a momentary pricing plan of Amazon EC2. Cost reduction is relative to level 1 compression. The optimal level, presented as percentage of sites where each level is the optimal

4 Implementation

Our main contribution to compression automation is software/infrastructure implementations that endow existing web servers with the capability to monitor and control the compression effort and utility. The main idea of the implemented compression-optimization framework is to adapt the compression effort to quality-parameters and the instantaneous load at the server. This adaptation is carried out fast enough to accommodate rapid changes in demand, occurring in short time scales of seconds.

In this section we provide the details of two alternative implementations and discuss their properties: Both alternatives are found in the project's site [36] and are offered for free use and modification:

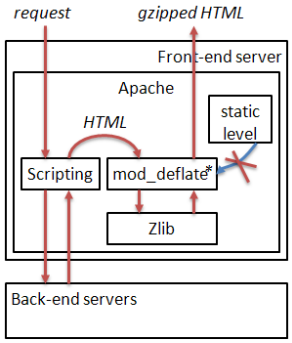
System-A - A modification of the deflate module of Apache (mod_deflate) that adapts the compression level to the instantaneous CPU load at the server.

System-B - Two separate modules work in parallel to offer a flexible but more complex solution: a monitor-and-set stand-alone process and an enhancement plugin for the compression entity.

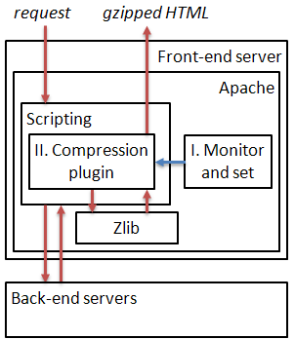
4.1 System-A

System-A is a transparent solution designed for seamless deployment and use in any working Linux-Apache environment. It does not require any changes in the working code and site structure, but requires a build of Apache with a modified *mod_deflate.c*. The modified module contains 100+ new lines of code in C language.

Figure 6a illustrates the system's architecture, which is very similar to a standard Apache. The only difference between a standard Apache and System-A, is that the *mod_deflate* module does not use the static compression level from the configuration file. Instead, the mod-



(a) System-A: Apache with a modified mod_deflate module



(b) System-B: monitor-and-set and a plugin

Figure 6: Implementation architectures

ule performs the following: 1) continuously checks the system's load, 2) remembers what was the last compression level in use, and 3) updates periodically the compression level, if needed.

The effort adaptation is performed in one-step increments and decrements. Proc. 1 gives a short pseudo-code that presents the basic idea in a straight-forward manner.

Proc. 1 Simplified pseudo-code of the level modification phase in *mod_deflate.c*

```
1. if next_update > cur_time then
2.     if cpu > high_threshold and cur_level > 1 then
3.         cur_level ← cur_level − 1
4.     else if cpu < low_threshold and cur_level < 9
       then
5.         cur_level ← cur_level + 1
6.     end if
7.     next_update ← cur_time + update_interval
8. end if
```

4.2 System-B

System-B takes a different approach – it is agnostic to the OS and web-server type, but it requires a change in the original site's code. Two separate modules in our implementation work in parallel to offer fast-adapting compression configuration. These are:

Monitor and set – Stand-alone process that monitors the instantaneous load incident on the machine, and chooses a compression setup to match the current machine conditions.

Compression plugin – Enhancement plugin for the compression entity that uses the chosen setup to compress the content. It operates in a fine granularity allowing to mix different setups among the outstanding requests.

Figure 6b illustrates the implementation structure. This solution compresses the HTML before it is being handed back from scripting to Apache. This allows the adaptive solution to bypass the static built-in compression mechanism and add the desired flexibility.

4.2.1 Module I – Monitor and Set

The monitor-and-set module runs as a background process on the same machine where compression is performed. Its function is essentially a control loop of the CPU load consisting of measuring the load and controlling it by setting the compression level. Its implementation as a separate process from the compression module is designed to ensure its operation even at high loads by proper prioritization.

4.2.2 Module II – Compression Plugin

The compression plugin is designed as an enhancement, rather than replacement, of an existing compression tool. This design allows our scheme to work with any compression tool chosen by the web-server operator or target platform. The sole assumption made about the compression tool in use is that it has multiple compression setups, ordered in non-descending effort levels. For example, the widely used zlib [13] compression tool offers a sequence of 9 setups with generally increasing effort levels and non-increasing compressed sizes.

This plugin code should be added at the end of an existing script code (like PHP), when the content is ready for final processing by the web-server. The plugin uses the setup provided by the monitor-and-set process and invokes the platform's compression tool with the designated setup. An important novelty of this algorithm is its ability to implement a non-integer setup number by mixing two setups in parallel. The fractional part of the setup

number determines the proportion of requests to compress in each of the integer setups. For example, when the input setup number is 6.2, then 80% of the requests can be compressed with setup 6, while the rest are compressed with setup 7.

4.3 Practical Considerations

We now turn to discuss the details pertaining to the specific implementations we use in the evaluation. Part of the discussion will include special considerations when operating in a cloud environment. Fully functional versions of this code, for different environments, can be obtained from the project's web page [36].

CPU monitoring uses platform-dependent system tools, and further needs to take into account the underlying virtual environment, like the Amazon EC2 we use in the paper. Specifically, we need to make sure that the CPU load readout is an accurate estimate [19] of the processing power available for better compression in the virtual server. From experimentation within the Amazon EC2 environment, we conclude that Linux utilities give accurate CPU utilization readouts up to a cap of CPU utilization budget determined by the purchased instance size. For example, in an m1.small instance the maximal available CPU is about 40%, while m1.medium provides up to 80% CPU utilization. These CPU budgets are taken into account in the CPU threshold parameters.

When deploying in a real system, the compression level should not be adapted too frequently. To ensure a graceful variation in CPU load, it is advisable to choose a minimal interval of at least 0.5 second.

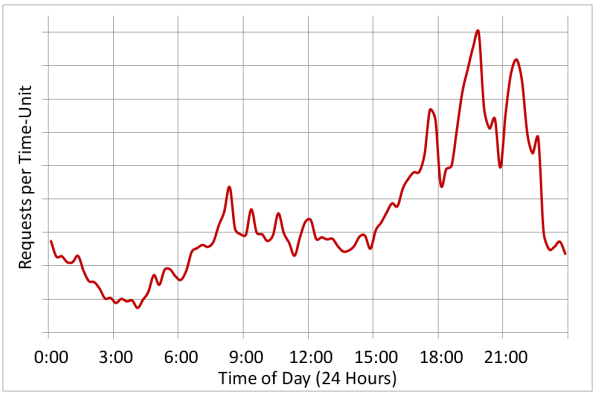
5 Proof of Concept Scenarios

After detailing our implementation of elastic web compression, in this section we turn to report on usage scenarios and our experience with the code. For the study we choose the Amazon EC2 environment, for both its popularity and flexibility. In the study we compare, under different scenarios of interest, the performance of our implementation of elastic compression to the static compression currently employed in popular commercial web-servers.

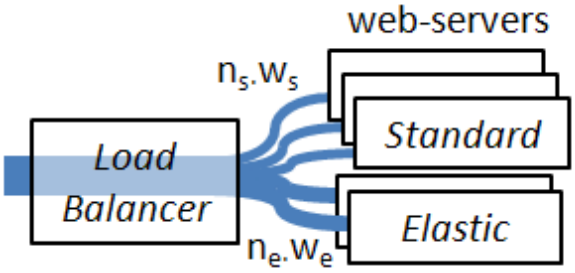
Our implementation is next shown to offer the following features:

1. Protecting against sudden demand peaks and denial-of-service (DoS) attacks.
2. Trading server's free CPU time for bandwidth savings at low/medium loads.

Before dispatching to these individual scenarios, we describe the general setup of our experiments.



(a) Workload



(b) System

Figure 7: Experiment setup - the workload is divided between a collection of standard web-servers and servers that run our elastic compression.

5.1 General Setup

The workload and the general setup are illustrated in Figure 7. We use EC2 instances of type m1.small, each providing 1.7 GB memory and 1 EC2 Compute Unit [3]. Each instance is a single front-end web-server, with Ubuntu Server 12.04.2 LTS 64-bit, Apache 2.2.24 (Amazon) and PHP 5.3.23. For a side-by-side comparison with standard web-servers ("Standard"), we equipped one or more of the servers with our elastic implementation ("Elastic"). Monitoring of the instances' resources and billing status is performed with Amazon CloudWatch.

The workload is a 24-hour recording of 27,000 distinct users who visited a specific social network site. This workload is typical to sites of its kind, demonstrating low demand at 5:00 AM and high peaks around 8:00 PM, as illustrated in Figure 7a. We replay this workload by running multiple clients from multiple machines in AWS. Each front-end web-server receives a fraction of the requests in any given time, through a load-balancer. The fractions are depicted in Figure 7b as w_s for each "Standard" server, and w_e for each "Elastic" server. More details are provided in each scenario separately.

5.2 Case 1: Spike/DoS Protection

In this sub-section we use the elastic compression as a protection utility against sudden traffic peaks and/or organized DoS attacks.

The "Elastic" maximal deflate level is set to 6, which is the default and the most popular compression setup of Apache (gzip level 6), as we show in Section 3. All the web-servers share the overall traffic evenly, while one web-server is equipped with the elastic compression. To emulate a sudden peak or attack, we run the workload almost normally, with one exception: on 19:40-19:45 the service experiences a sudden 5 minutes attack, equivalent to 69% additional requests comparing with a normal high time-of-day. Figure 8a gives a zoom-in to the access pattern of a 1 hour interval containing the 5-minutes attack.

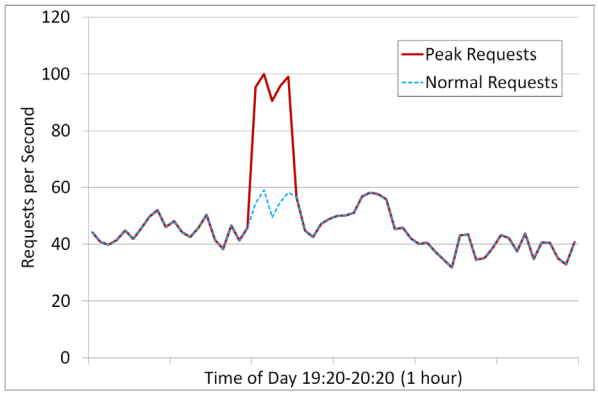
Figure 8b illustrates the projected latency at the clients who managed to complete a request, showing that the static server is saturated during the attack and also long after it. In addition (not shown in graph), almost half the client requests timed-out during the 5-minutes peak. Practically, all the "Standard" servers were out of order for at least 10 minutes. In reality, these attacks usually end up worse than that, due to client retransmission and/or servers that fail to recover. Figure 8c shows how the elastic compression handles the attack: it lowers the compression effort to minimum during the attack, until it feels that the server is no longer in distress.

5.3 Case 2: Compress More

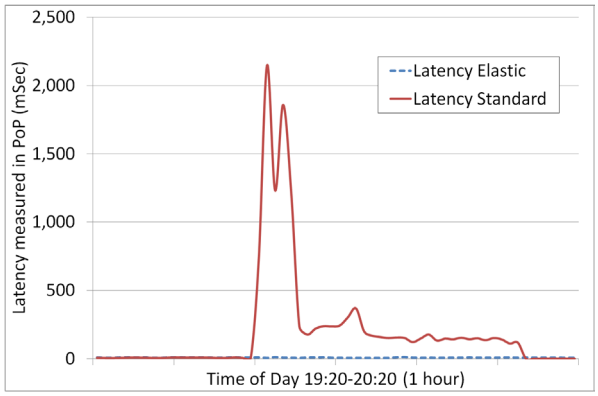
We consider a standard web-server running low-effort compression. More specifically, the baseline is "Standard" Apache servers running compression level 1 (a.k.a fastest).

In this experiment, all the servers receive an equal share of the traffic throughout the day – meaning $w_s = w_e$. While a "Standard" server uses the same compression setup all day long, an "Elastic" server selectively changes the compression level: from 1 (fastest) to 9 (slowest). It changes the compression level according to the sensed load at the server, in order to compress more than the standard server when the end-users' demand is relatively low.

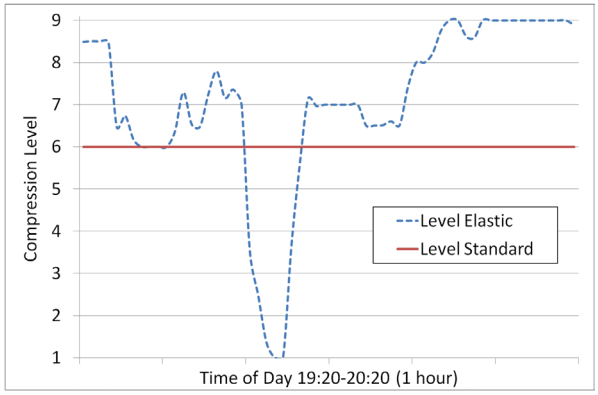
The standard machine's CPU follows the end-users' request pattern tightly, leaving large portion of the CPU unused most of the day. The elastic solution uses the free CPU for higher-effort compression most of the day, saving additional bandwidth, whenever there are enough unused resources for it. When demand is high, the elastic solution returns to low-effort compression, staying below the maximal allowed CPU consumption. In the given scenario, the adaptive compression level managed to save 10.62% of the total traffic volume during the 24-hours experiment.



(a) Requests - additional 69% in 5 minutes interval



(b) High latency in the standard server during the peak



(c) Elastic compression level, as used in practice

Figure 8: Case 3: handling a sudden peak and/or DoS attack.

5.4 Case 3: Reducing the 95th Percentile Latency

We now evaluate elastic compression on the real-world workload of another big web infrastructure[1]. While serv-

[1]The source has asked to remain anonymous, keeping its systems' structure and performance hidden from hostiles

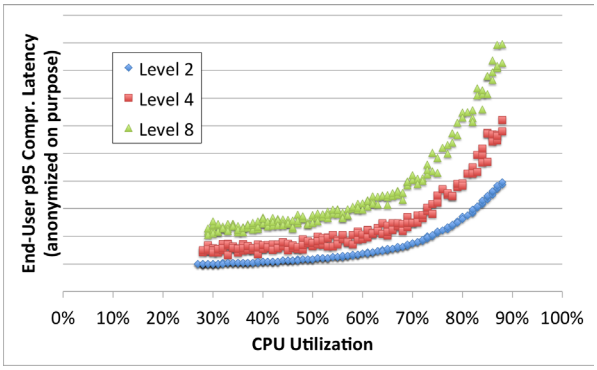


Figure 9: Reducing p95 latency: normalized compression times at different levels as a function of CPU utilization.

ing millions requests a second, its web servers strive to reduce response generation times for providing optimal user experience. Therefore, a compression strategy in such an environment should be as lightweight as possible while trying to minimize the needed bandwidth.

Measuring the latency of active real-world services is tricky, because the service provider is mostly worried about loosing customers that experience the worst latencies. Hence, it is a common practice to consider percentiles and not just the average. As compression times vary heavily among different responses we consider percentile p95 a representative of "heavy compressions".

In the following experiment we compare compression times and response sizes of levels 2, 4 and 8 as a function of CPU utilization at a web server. Figure 9 demonstrates that the differences between compression times at different levels grow super linearly as a function of CPU utilization[2]. This presents an opportunity to sacrifice 10% of the egress bandwidth under heavy load in order to drastically (x2) reduce the compression time.

6 Background and Related Work

Data compression and its effect on computer systems is an important and well studied problem in the literature. Here we survey some background on compression tools, and related work on compression-related studies.

It is important to first note that the popular compression term gzip stands for several different things: a software application [16], a file format [8, 9], and an HTTP compression scheme. Gzip uses the *deflate* compression algorithm, which is a combination of the LZ77 algorithm [35] and Huffman coding. Gzip was officially adopted by the web community in the HTTP/1.1 specifications [11]. The standard allows a browser to declare its gzip-decompression capability by sending the server an HTTP request field in the form of "Accept-Encoding: gzip". Gzip is the most broadly supported compression method as of today, both by browsers and by servers. The server, if supports gzip for the requested file, sends a compressed version of the file, prefixed by an HTTP response header that indicates that the returned file is compressed. All popular servers have built-in support or external modules for gzip compression. For example, Apache offers mod_deflate (in gzip format, despite the misleading name) and Microsoft IIS and nginx have built-in support.

It became a standard in the major compression tools to offer an effort-adjustment parameter that allows the user to trade CPU for bandwidth savings. In gzip the main parameter is called *compression level*, which is a number in the range of 1 ("fastest") to 9 ("slowest"). Lower compression levels result in a faster operation, but compromise for size. Higher levels result in a better compression, but slower operation. The default level provides an accepted compromise between compression ratio and speed, and is equivalent to compression level 6.

Industrial solutions for gzip include PCI-family boards [22, 1], and web accelerator boxes [10] that offload the CPU-intensive compression from the servers. Although these solutions work in relatively high speeds, they induce additional costs on the website owner. These costs make the hardware solutions inadequate for websites that choose compression for cost reduction in the first place.

There is an extensive research on compression performance in the context of energy-awareness in both wireless [4] and server [23] environments. Inline compression decision [6] presents an energy-aware algorithm for Hadoop that answers the "to compress or to not compress" question per MapReduce job. Similarly, fast filtering for storage systems [17] quickly evaluates the compressibility of real-time data, before writing it to storage. Fine-grain adaptive compression [30] mixes compressed and uncompressed packets in attempt to optimize throughput when the CPU is optionally the bottleneck in the system. A more recent paper [15] extends the mixing idea to scale down the degree of compression of a single document, using a novel implementation of a parallelized compression tool. A cloud related adaptive compression for non-web traffic [20] focuses on how to deal with system-metric inaccuracy in virtualized environments. We found that this inaccuracy problem, reported in July 2011 [19], no longer exists in Amazon EC2 today. To the best of our knowledge, our paper presents the first adaptive web compression that takes a complex cost-aware decision with multiple documents.

In this paper we use the gzip format and zlib [13] software library to demonstrate our algorithms, because

[2]The Y-axis values were normalized/anonymized on purpose, per the source's request.

both are considered the standard for HTTP compression. Nevertheless, the presented algorithms are not limited to any particular compression technology or algorithm; they can encompass many different compression algorithms and formats. In particular, they can also benefit from inline compression techniques [24, 33, 26] when support for these formats is added to web servers and browsers.

7 Conclusions

In this paper we had laid out a working framework for automatic web compression configuration. The benefits of this framework were demonstrated in several important scenarios over real-world environments. We believe that this initial work opens a wide space for future research on compression cost optimization in various platforms, including cloud-based services. Building on the main functionality of our proposed implementation, future implementations and algorithms can improve cost by tailoring compression to more system architecture and content characteristics.

Acknowledgements

We would like to thank Alexander Tzigirintzev and Yevgeniy Sabin for their implementation of System-A project presented above, and Roy Mitrany for his invaluable feedback and insight in this project.

The authors would like to thank the anonymous LISA 2014 reviewers and our shepherd, Matt Simmons, for their comments and suggestions that considerably helped us to improve the final version.

This work was supported in part by a European Union CIG grant, and by the Israeli Ministry of Science and Technology.

Availability

Our code and projects mentioned in this paper, are free software, available on a web site, along with additional information and implementation details, at

`http://eyalzo.com/ecomp`

References

[1] AHA GZIP Compression/Decompression Accelerator. `http://www.aha.com/index.php/products-2/data-compression/gzip/` .

[2] Alexa Internat, Top Sites. 1996. `http://www.alexa.com/topsites` .

[3] Amazon EC2 Instance Types. April 2013. `http://aws.amazon.com/ec2/instance-types/` .

[4] K. Barr and K. Asanović. Energy Aware Lossless Data Compression. *Proc. of MobiSys*, 2003.

[5] A. Bremler-Barr, S. T. David, D. Hay, and Y. Koral. Decompression-Free Inspection: DPI for Shared Dictionary Compression over HTTP. In *Proc. of INFOCOM*, 2012.

[6] Y. Chen, A. Ganapathi, and R. H. Katz. To Compress or Not to Compress - Compute vs. IO Tradeoffs for MapReduce Energy Efficiency. In *Proc. of SIGCOMM Workshop on Green Networking*, 2010.

[7] Y. Chess, J. Hellerstein, S. Parekh, and J. Bigus. Managing web server performance with autotune agents. *IBM Systems Journal*, 42(1):136–149, 2003.

[8] P. Deutsch. DEFLATE Compressed Data Format Specification Version 1.3. *RFC 1951*, May 1996.

[9] P. Deutsch. GZIP File Format Specification Version 4.3. *RFC 1952*, May 1996.

[10] F5 WebAccelerator. `http://www.f5.com/products/big-ip/big-ip-webaccelerator/` .

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. *RFC 2616*, June 1999.

[12] W. Foundation. Web page performance test. 2014. `http://www.webpagetest.org` .

[13] J. L. Gailly and M. Adler. Zlib Library. 1995. `http://www.zlib.net/` .

[14] J. Graham-Cumming. Stop worrying about time to first byte (ttfb). 2012. `http://blog.cloudflare.com/ttfb-time-to-first-byte-considered-meaningles` .

[15] M. Gray, P. Peterson, and P. Reiher. Scaling Down Off-the-Shelf Data Compression: Backwards-Compatible Fine-Grain Mixing. In *Proc. of ICDCS*, 2012.

[16] Gzip Utility. 1992. `http://www.gzip.org/` .

[17] D. Harnik, R. Kat, D. Sotnikov, A. Traeger, and O. Margalit. To Zip or Not to Zip: Effective Resource Usage for Real-Time Compression. In *Proc. of FAST*, 2013.

[18] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart. Next stop, the cloud: Understanding modern web service deployment in ec2 and azure. In *Proc. of IMC*, 2013.

[19] M. Hovestadt, O. Kao, A. Kliem, and D. Warneke. Evaluating Adaptive Compression to Mitigate the Effects of Shared I/O in Clouds. In *Proc. of IPDPS Workshop*, 2011.

[20] M. Hovestadt, O. Kao, A. Kliem, and D. Warneke. Adaptive Online Compression in Clouds - Making Informed Decisions in Virtual Machine Environments. *Springer Journal of Grid Computing*, 2013.

[21] IANA - HTTP parameters. May 2013. `http://www.iana.org/assignments/http-parameters/` .

[22] Indra Networks PCI Boards. `http://www.indranetworks.com/products.html` .

[23] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok. Energy and Performance Evaluation of Lossless File Data Compression on Server Systems. In *Proc. of SYSTOR*, 2009.

[24] R. Lenhardt and J. Alakuijala. Gipfeli-High Speed Compression Algorithm. In *Proc. of Data Compression Conference (DCC)*, 2012.

[25] Z. Li, D. Levy, S. Chen, and J. Zic. Explicitly controlling the fair service for busy web servers. In *Proc. of ASWEC*, 2007.

[26] LZO Data Compression Library. `http://www.oberhumer.com/opensource/lzo/` .

[27] Microsoft IIS - Dynamic Caching and Compression. `http://www.iis.net/overview/reliability/dynamiccachingandcompression` .

[28] Netcraft Web Server Survey. November 2012. `http://news.netcraft.com/archives/2012/11/01/november-2012-web-server-survey.html` .

[29] S. Pierzchala. Compressing Web Content with mod_gzip and mod_deflate. In *LINUX Journal*, April 2004. `http://www.linuxjournal.com/article/6802` .

[30] C. Pu and L. Singaravelu. Fine-Grain Adaptive Compression in Dynamically Variable Networks. In *Proc. of ICDCS*, 2005.

[31] A. Ranjan, R. Kumar, and J. Dhar. A Comparative Study between Dynamic Web Scripting Languages. In *Proc. of ICDEM*, pages 288–295, 2012.

[32] S. Schwartzberg and A. L. Couch. Experience in Implementing an HTTP Service Closure. In *LISA*, 2004.

[33] snappy Project - A Fast Compressor/Decompressor. `http://code.google.com/p/snappy/` .

[34] S. Souders. High-performance web sites. *Commun. ACM*, 51(12):36–41, Dec. 2008.

[35] J. Ziv and A. Lempel. A Universal Algorithm for Sequential data Compression. *Information Theory, IEEE Transactions on*, 1977.

[36] E. Zohar and Y. Cassuto. Elastic compression project homepage. 2013. `http://www.eyalzo.com/projects/ecomp/` .