Empirical Study on Effects of Script Minification and HTTP Compression for Traffic Reduction

Yasutaka Sakamoto, Shinsuke Matsumoto, Seiki Tokunaga, Sachio Saiki and Masahide Nakamura
Graduate School of System Informatics, Kobe University
1-1 Rokkodai, Nada, Kobe, Hyogo 657-8501, Japan
Email: gen@ws.cs.kobe-u.ac.jp, shinsuke@cs.kobe-u.ac.jp

***Abstract*—Code-on-demand is an architectural style that a client dynamically downloads a raw script file and executes it on the client-side. This style causes a problem of network traffic because a raw script is not always compiled or minified in advance. Formatting rules, such as indents, line breaks and comments for ensuring human readability, are not necessary to the execution. In order to save wasteful data transfer, it is necessary to minify or optimize the script on the entirety of the Web. In this paper, we explore the potential for JavaScript size reduction with focus on the two reduction approaches: script minification and HTTP compression. The main two research questions are: *RQ1: How many percent of websites have reduction potential? RQ2: How much JavaScript size can be reduced on the Web?* Our results show that about 40% of total size of JavaScript files used on the top 500 websites can be potentially reduced by a script minification. Moreover, the current JavaScript data traffic is saving over 50% by HTTP compression. If every website was configured to use HTTP compression, we can achieve a reduction rate of 5% to 20%.**

***Keywords*—Traffic reduction, script optimization, HTTP compression, code-on-demand, JavaScript**

I. Introduction

With the advent of HTML5 and the emerging Web technologies, Web browser is becoming like an operating system [1][2]. In the following such kind of Web-centered computing environment, large variety of software capabilities are distributed on the Web, and are connected with each other by JavaScript [3]. Roles of the JavaScript are extending far beyond XML/HTML DOM manipulation. Brief examples of the extended roles are the following: constructing a rich user interface, mashup of existing Web APIs, real-time messaging between client and server, current position retrieval, etc.

Every scripting languages, which are executed on Web browser (e.g., JavaScript, ActionScript and Java applets), have a distinct feature named *code-on-demand* (also called *Client-side scripting*). This is one of a constraint of the architectural style of REST (Representational State Transfer) [4]. According to the code-on-demand style, a client (i.e., Web browser) dynamically downloads a raw script file and executes it on the client-side. The architectural style that a client downloads and executes a pre-compiled executable binary is an opposite style of the code-on-demand. Code-on-demand allows flexible update of client-side capabilities after deploying a Web application without any modification of a server-side program.

However, code-on-demand causes a problem of network traffic. One reason is that since a raw script is downloaded to a client, it is not always compiled or minified in advance of deployment. Formatting rules, such as indents, line breaks and comments for ensuring human readability, are not necessary to the execution. Moreover, annoying wasteful logical statements, such as unreachable code, redundant code and unused code, are potentially hidden in every script. They should be omitted or eliminated to save wasteful data transfer. Another reason is in the HTTP protocol. Essentially, text-based Web contents (e.g., HTML, CSS and JavaScript) have a high potential for data compression. Although HTTP/1.1 defines a capability of transferred data compression, it is disabled by default in the common Web server because the compression and decompression cause a minor performance hit to both server-side and client-side.

In order to save wasteful data transfer, it is necessary to minify or optimize a script on the entirety of the Web. Extremely high-speed network infrastructures are becoming more and more popular in developed countries. However, we still face crowded and low-speed Wi-Fi environments on airport, cafe, international conference, etc. Especially, a network environment of mobile devices requires efficient usage of network bandwidth.

Multimedia contents are occupying a lot of the current Web traffic (about half, according to [5]) because their file size is basically large compared to text-based Web contents. However, applying an encoding or compression process, whether lossy or lossless, for the multimedia contents was already common practice. In contrast, size reduction techniques (e.g., minification, optimization and compression) for text-based contents are often neglected due to the its small size. Hence, we assume that the existing text-based Web contents still have certain amount of reduction potential.

Our long-term goal of this research is to conduct an extensive empirical study of *potential for size reduction of existing text-based Web contents*. In this paper, we focus on the reduction potential of JavaScript files in terms of the code-on-demand style. Two approaches are studied as a technique of JavaScript size reduction: script minification [6][7] and HTTP compression [8][9]. Script minification is a technique that reduces size of script files by omitting and eliminating unnecessary codes. HTTP compression allows Web contents to be compressed on server-side before transferring to a client.

ISBN: 978-1-4799-6375-1 ©2015 IEEE

The main two research questions are formulated as follows.
RQ1: *How many percent of websites have reduction potential?*
RQ2: *How much JavaScript size can be reduced on the Web?*

II. Reducing JavaScript Size

A. Definition of Terms and Measures

We first define the following two terms.

Processed or not: The term "Processed" means a given JavaScript file is already applied script minification or HTTP compression. Deciding whether a given website is supporting HTTP compression is simple and clear task because HTTP response header includes it. However, deciding whether a script is already minified is not an easy task because of a variety of minification strategies. In this paper, we regard a given script as not processed (i.e., not minified) if the script includes one or more indents or comments. This confirmation process is conducted by manually.

Reduction rate: This metric is a key indicator in our empirical study. The reduction rate represents how much can JavaScript size be reduced by script minification or HTTP compression. In this paper, we formulate the reduction rate as follow.

$$ReductionRate(\%) = 100 \times (1 - \frac{ProcessedSize}{UnprocessedSize}) \quad (1)$$

B. Script Minification ($Script_{min}$)

Script minification is a technique to reduce script size without any modification on essential process. In this paper, we labeled this technique as $Script_{min}$.

Here, we describe a concrete example of $Script_{min}$. The following raw code snippet of JavaScript is an example of summing all integers from 0 to 10. The size of this script is 68 bytes.

```
var sum=0;
for (var i=0; i<=10; i++) {
  sum += i;
}
alert(sum);
```

The basic process of $Script_{min}$ is removing a variety of formatting rules such as indent and comment which are written for ensuring human readability.

```
var sum=0;for(var i=0;i<=10;i++){sum+=i;}aler
t(sum);
```

As shown in the above code snippet, every line breaks and unnecessary white-spaces are omitted. The reduction rate of script size is about 24% by this process.

The most powerful minification is based on considering logical behavior. This minification can be regarded as one aspect of *program optimization* [10]. For example, unused variable and unreachable code statement are unnecessary to the execution and can be omitted. The above example script can be minified as follows by using the logical optimization.

```
alert(55);
```

In the result, we achieve a reduction rate of 85% (i.e., 68 bytes are minified to 10 bytes).

Many variety of minification tools have been released on the Web. Not surprisingly, they have different features in terms of minification strategies, principles, and minification effects. In the experiment, we compare three minification tools to compare these features (see Section III-C).

C. HTTP Compression ($HTTP_{comp}$)

HTTP compression is an optional feature of HTTP/1.1 [4] to reduce the amount of HTTP traffic of Web contents [9]. It allows Web contents to be compressed on server-side before transferring to a client. In general, gzip format, which uses Lempel-Ziv (LZ77) algorithm with Huffman coding, is used as a compression algorithm. In this paper, we labeled this as $HTTP_{comp}$.

An example of a sequence of a $HTTP_{comp}$ negotiation is follows. First, a client sends a request message to a server with "Accept-Encoding" parameter in the request header to enable $HTTP_{comp}$.

```
GET /index.html HTTP/1.1
Host: example.com
Acccept-Encoding: gzip
```

Then, the server sends a response message with the following response header.

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
```

In this case, the requested and transferred HTML content (i.e., http://example.com/index.html) written in the response body is compressed by gzip. It is necessary for the client to decompress the response body before parsing the HTML content.

$HTTP_{comp}$ is supported by a common Web server software (e.g., Apache HTTP Server, Internet Information Services and Nginx). However, $HTTP_{comp}$ is disabled by default because it takes a minor performance hit on both client and server for the decompression. We consider that this disablement can be potential for JavaScript size reduction.

D. Pilot Study

In order to confirm the reduction effects of $Script_{min}$ and $HTTP_{comp}$, we have conducted a pilot study. The experimental objects are famous and widely used JavaScript libraries: jQuery (ver. 2.1.1), prototype.js (ver. 1.7.2) and backbone.js (ver.1.1.2). Though the three libraries are provided with an already-minified version on their website (basically suffixed with ".min.js"), we use a non-minified version. As noted in the Section II-B, there are a variety of minification tools. We picked up YUI compressor which minifies a script by omitting indents, comments, line breaks and by abbreviating local variable names.

Table I shows the result of the pilot study. Percentage in parentheses represents reduction rate. Script sizes are reduced

ISBN: 978-1-4799-6375-1 ©2015 IEEE

TABLE I. REDUCTION EFFECTS OF $Script_{min}$ AND $HTTP_{comp}$

Subject JS lib.	unprocessed	$Script_{min}$	$HTTP_{comp}$	Both
jQuery	241.6 KB	128.2 KB	72.9 KB	37.0 KB
	—	(47%)	(70%)	(85%)
prototype	193.1 KB	102.3 KB	45.3 KB	33.0 KB
	—	(47%)	(77%)	(83%)
backbone.js	59.6 KB	19.7 KB	17.3 KB	6.9 KB
	—	(67%)	(71%)	(88%)

about 30% to 50% for each JavaScript library by applying $Script_{min}$. In other words, $Script_{min}$ can reduce the JavaScript size to 50% to 70%. $HTTP_{comp}$ also have high reduction rate about 70%. This is because Huffman coding provides effective performance for a text-based file. Moreover, we can achieve 90% reduction rate by combining $Script_{min}$ and $HTTP_{comp}$. The reason is that the processing result of $Script_{min}$ is just a text-based JavaScript file and $HTTP_{comp}$ is effective to a text-based file. Therefore, $Script_{min}$ and $HTTP_{comp}$ can independently contribute to size reduction.

E. Research Questions

Our research questions can be formulated as follows.

RQ1: *How many percent of websites have reduction potential?*

We first study the percentage of websites, which have some possibilities for size reduction, before studying the actual reduction rate on the Web. The degree of the possibility can be regarded as one of characteristics of potential for JavaScript size reduction. To answer this question, we explore whether or not a JavaScript is already applied $Script_{min}$ and $HTTP_{comp}$.

RQ2: *How much JavaScript size can be reduced on the Web?*

The second research question is the most essential for this study. By crawling a lot of websites, we explore the *actual reduction rate* by using both $Script_{min}$ and $HTTP_{comp}$. If there still remain many JavaScript, which can be reduced, we have a potential to achieve more efficient bandwidth usage on the crowded and low-speed Wi-Fi environment. To answer this question, we crawl the top-ranking websites and retrieve all JavaScript files. Then, we apply $Script_{min}$ and $HTTP_{comp}$ to the all JavaScript files, and compare with unprocessed one.

III. EXPERIMENT DESIGN

A. Subject Website

We use a list of Alexa Top 500 Global Website[1] as an object of the study. The list of top ranking website is suitable to confirm our research questions that try to empirically explore the entirety of the Web. Especially, if a popular website has high reduction potential, it can be highly effective for reducing the amount of network traffic.

Table II represents summary of lists of subject websites. 22 of 500 websites was already not found and was ignored in our study. Focusing on Alexa top 500 shows every website includes about 20 separated JavaScript files in average. The total size of JavaScript files was 157 MB. This size can be regarded as a baseline measurement in the study. The answer of RQ2 indicates how the 157 MB can be reduced by $Script_{min}$ and $HTTP_{comp}$.

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

TABLE II. SUMMARY OF SUBJECT WEBSITE LIST

Metric	Subject website list	
	Alexa top 500	Japanese gov. univ.
# included websites	500	86
# not found websites	22	0
average # external JS files	5.8	6.3
average # internal JS files	15.5	5.7
average # JS files	21.3	12.0
average size of JS files	314 KB	141 KB
total size of JS files	157 MB	12 MB

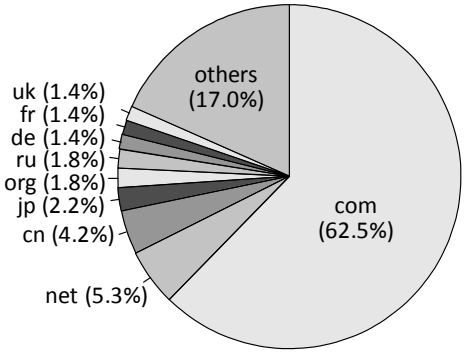


Fig. 1. Percentage of top-level domain of Alexa top 500 websites

Note that the website list has some bias because the Alexa's ranking data is collected through Alexa Toolbar which is installed on the user's browser. So, the bias and the results are influenced by the population of users of the Alexa Toolbar. To confirm the dataset bias, we show percentages of top-level domain of the top 500 websites in Figure 1. About 70 % websites are belong to "com", "net" or "org" (i.e., commercial organizations, network infrastructures or organizations). Moreover, there are few country code top-level domains. It includes East and North Asia (China, Japan and Russia), and Europe (Germany, France and United Kingdom). This distribution may strongly affect to our empirical results.

In addition, we use a list of websites of Japanese all 86 government universities. The current World Wide Web includes an enormous number of websites and is growing more and more. It is necessary to explore not only popular websites but also various other non-popular websites in order to answer our questions. Therefore, we chose Japanese universities as an example non-popular websites.

Note that our study explores only the root directory (i.e., welcome index page) of a specified website because we assumed that the welcome page has a strong tendency to reflect other web contents in the same website.

B. Experimental Procedure

Figure 2 shows an experimental procedure for both RQ1 and RQ2.

1. *Fetching to root domain:* For a given domain name ("example.com" in the figure), we send an HTTP GET request to the root directory and retrieve a root index page.
2. *JavaScript extraction:* The response body is analyzed by an HTML DOM parser to extract scripts.

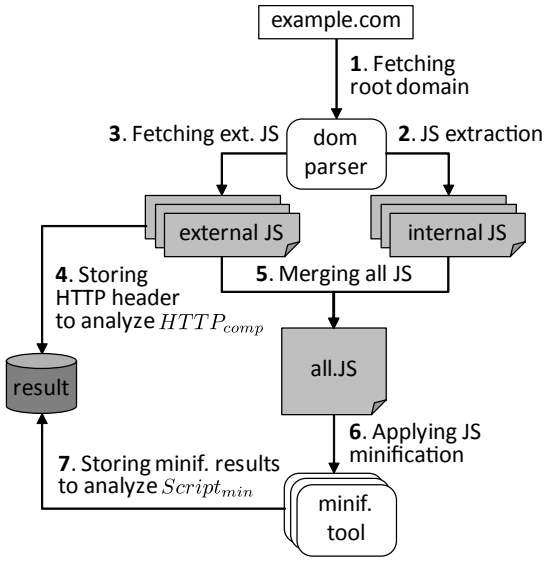


Fig. 2. Experimental Procedure

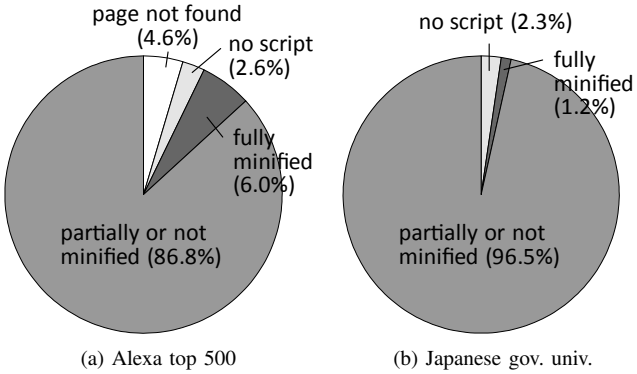
Scripts written directly in an HTML document are labeled "internal JavaScript". They are described within `<script>` tag.

3. *Fetching external JavaScript:* The HTML document includes some "external JavaScript" which are dynamically loaded on client-side. They are fetched respectively. The URI of the external JavaScript is described as `<script src="ext.js">`.
4. *Storing HTTP header to analyze $HTTP_{comp}$:* An HTTP response header which is retrieved on Step.3 is stored to a database. This header includes information whether the external JavaScript is transferred with HTTP compression or not.
5. *Merging all JavaScript:* Internal and external JavaScript are merged with holding the order of execution.
6. *Applying JavaScript minification:* The "all.js" is minified by using three minification tools. More detailed description of these tools are shown in the next section.
7. *Storing minification results to analyze $Script_{min}$:* The effects of script minification, such as reduction rate or minification time, are saved to analyze $Script_{min}$.

C. Minification Tools

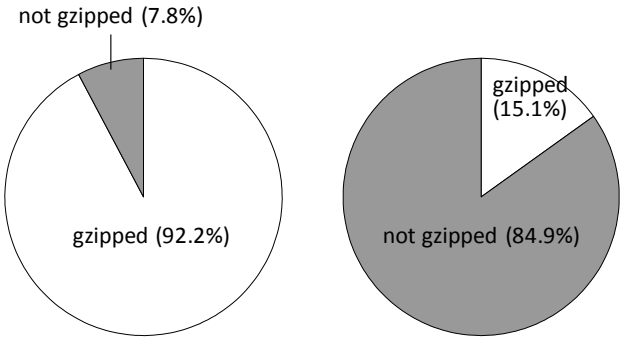
We selected the following three tools for the empirical study from a variety of minification tools published on the Web. We assumed that the first (YUI) has high-applicability but less-reduction rate, in contrast, the second (CC) has high-reduction rate but low-applicability, and the third (packer) is an obfuscation tool.

YUI Compressor [11] (abbr. YUI) is a well-known tool for JavaScript minification. YUI supports some format minification (e.g., omitting white-spaces, indents, and line breaks) and a few logical optimization (e.g., shrinking variable names).



(a) Alexa top 500 (b) Japanese gov. univ.

Fig. 3. Percentages of the usage of $Script_{min}$



(a) Alexa top 500 (b) Japanese gov. univ.

Fig. 4. Percentages of the usage of $HTTP_{comp}$

The advantage of this tool is less side effects on script behavior compared with other minification tools. This paper assumes this tool as a baseline of script minification.

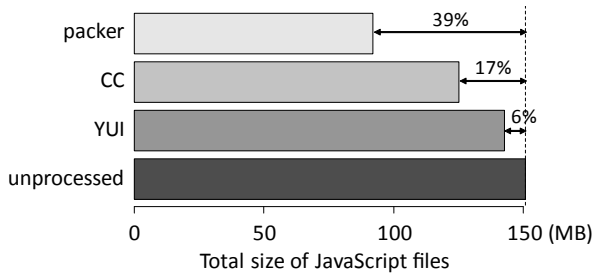
Closure Compiler [12] (abbr. CC) is one of a minification and optimization tool provided by Google. In contrast to YUI Compressor, CC supports powerful logical optimization strategies. For example, the most minified result described at Section II-B is processed by CC. However, CC imposes some restrictions on an optimized script in order to retrieve a maximally-optimized code. We assumed that CC has high efficiency but low-applicability for script minification.

Dean Edwards' JavaScript packer [13] (abbr. packer) is an obfuscation tool of JavaScript. In general, obfuscation drastically changes script logics to reduce readability. An obfuscated script by packer with Base62 encoding option is always started with as follows.

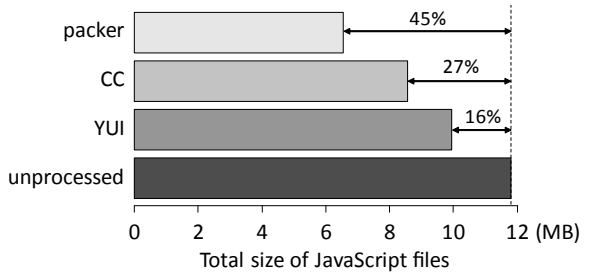
```
eval(function(p,a,c,k,e,d){...
```

Obfuscation and minification are similar but quite different in terms of the purpose and principles. However, obfuscation, particularly packer, also provides an similar effect of script minification. So, we picked up this tool as one of a studied minification tools.

All the above three tools have some optional parameters to select several algorithms for minification, optimization and obfuscation. The tool parameters used in the study are footnoted



(a) Alexa top 500



(b) Japanese gov. univ.

Fig. 5. Minification effects of $Script_{min}$

below[2].

IV. Experiment Result

A. RQ1: How many percent of websites have reduction potential?

First, we explain the result of $Script_{min}$. Figure 3 shows the percentages of the usage of JavaScript minification for Alexa top 500 websites (Figure 3a) and Japanese university websites (Figure 3b). The "partially or not minified" represents that a website has a certain amount of reduction potential because one or more scripts can be minified. The results show that the most (86.8%) of the top 500 websites have a potential for size reduction by using $Script_{min}$. In particular, non-popular websites, compared with the top 500 (i.e., Japanese university), have a high potential (96.5%).

Next, the result of $HTTP_{comp}$ is shown in Figure 4. This figure shows how many percentages of websites have been configured to support HTTP compression. From Figure 4a, we can see that the most of the popular websites (92.2%) already used $HTTP_{comp}$. On the other hand, Figure 4b shows that there are still remain many websites which do not support $HTTP_{comp}$.

B. RQ2: How much JavaScript size can be reduced on the Web?

Figure 5 shows the effect of minification tools and Figure 6 shows more detailed minification effects for the top 10 websites. The undermost "unprocessed" in Figure 5 represents the total size of the current JavaScript files for all specified websites. Please note that the "unprocessed" includes already minified scripts and raw (i.e., neither minified nor optimized) scripts. The other labels, "YUI", "CC" and "packer", represent the total size after applying each minification tool. The percentage means the reduction rate.

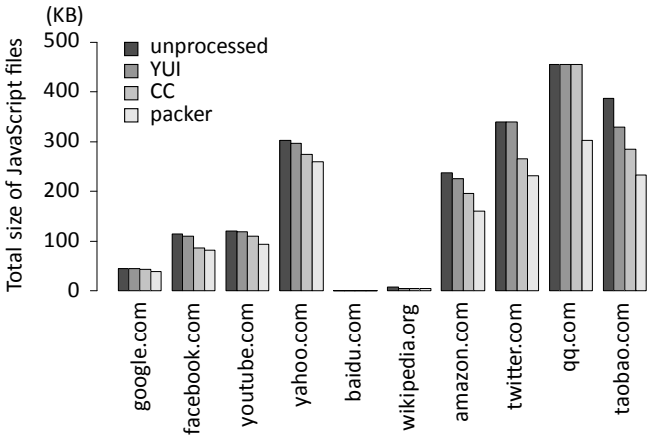


Fig. 6. Detailed minification effects for the top 10 websites

Figure 5 shows that Dean's packer is the highest reduction rate and YUI is the lowest. This is because packer has a quite different strategy from the other minification tools. More specifically, packer encodes a script based on Huffman coding algorithm on the supposition that the script is just a string data, not a program. Therefore, though the other tools have less effect on already minified scripts, packer is effective for whether already minified or not. This result also can be shown in Figure 6. So, we conclude that we can achieve a reduction rate about 39% by applying packer.

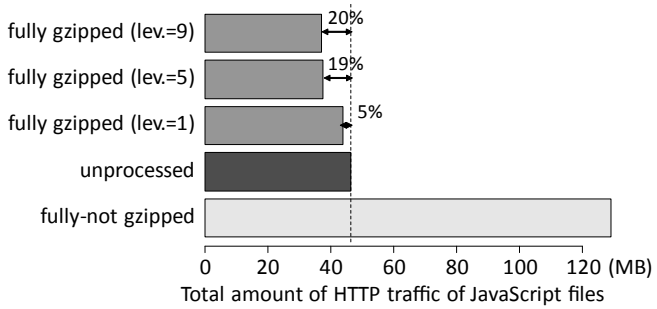
The effects of $HTTP_{comp}$ are shown in Figure 7. The "unprocessed" represents that we do not apply any process for captured HTTP traffic as with Figure 5. The undermost "fully-not gzipped" means a hypothetical situation where all the websites do not configure $HTTP_{comp}$. In contrast to that, "fully-gzipped" means a hypothetical situation where all the websites use $HTTP_{comp}$. The "lev." means a compression level, one is the lowest compression, nine is the best and five is the middle.

Focusing on the "fully-not gzipped" in Figure 7a shows the traffic size of JavaScript files on the current Web has already been reduced by more than half by $HTTP_{comp}$. If every website was configured to use $HTTP_{comp}$, we can achieve a reduction rate of 5% to 19%. Moreover, Figure 7b shows that Japanese university's websites have much further reduction rate (59% to 65%). So, we conclude that there still remain certain possibilities for the traffic size reduction by $HTTP_{comp}$.

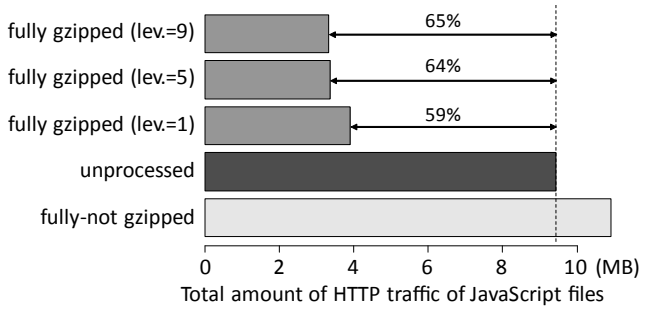
C. Related Works

The expected effects of applying $Script_{min}$ and $HTTP_{comp}$ for Web traffic reduction are strongly depends on the proportion of total size of JavaScript in all Web content. Ihm and Pai reported an analysis of modern Web traffic for

[2]**YUI Compressor:** Default option, which means the most powerful minification in YUI, is used. It allows omitting format style and replacing local symbols. **Closure Compiler:** Advanced optimization, the most aggressive setting, is selected. It transforms over the entire code with considering logical behavior. CC gives a compile error message if the target code does not keep compilation restrictions **packer:** Base62 encoding option is selected. Words are encoded into alphanumeric characters (a-z, A-Z and 0-9) to decrease code readability.



(a) Alexa top 500



(b) Japanese gov. univ.

Fig. 7. Compression effects of $HTTP_{comp}$

five years of real traffic data [5]. They found that multimedia contents consume about 50% (image and video are 40% and 10% respectively), text-based contents take about 45% (HTML, CSS and JavaScript are 25%, 5% and 15%) and others are 5%. Hence, we assume that our results would affect to the 15% of Web traffic data.

There are wide variety of techniques of Web contents minification and optimization [6][7][14][15][16]. Martin et al. have proposed another JavaScript compression tool, which reduces the size of JavaScript file by converting it into an abstract syntax tree [14]. The concept of minification, which reduces data size by omitting unnecessary code, can be applied for HTML and CSS [15]. Merging some Web contents into a single HTML file can save data transfer by eliminating some HTTP requests [6].

Chrome mobile browser is providing a browser embedded feature for reducing data traffic [17]. Turning on the "Reduce data usage" setting bypasses all HTTP requests to proxy servers hosted at Google. Text-based Web contents are compressed by simple minification process, and applied HTTP compression. Furthermore, image files are transcoded to WebP, which is an image format of lossy and lossless compression developed by Google. According to Google's report, this setting can reduce the size of web pages by 50%.

In contrast to the existing minification tools and articles, the contribution of our study is to examine and show the *reduction potential for real 500 websites*. Conducting empirical study with other minification techniques are our future work.

V. Conclusion

We have explored the potential for JavaScript size reduction with focus on the two approaches: script minification and HTTP compression. Our results raise following conclusions.

- 87% of top 500 websites have a certain amount of reduction potential by script minification.
- 92% of top 500 websites have already been configured to support HTTP compression, whereas most non-popular websites (85%) have not yet.
- 39% of the total size of JavaScript files can be potentially reduced by applying script minification.
- HTTP compression is saving over 50% of the current traffic size of total JavaScript files. If every website was configured to use HTTP compression, we can save a further 5% to 20% of JavaScript traffic.

Acknowledgment

This research was partially supported by the Japan Ministry of Education, Science, Sports, and Culture [Grant-in-Aid for Scientific Research (C) (No.24500079, No.24500258), Scientific Research (B) (No.26280115), Young Scientists (B) (No.26730155)] and Kawanishi Memorial ShinMaywa Education Foundation.

References

[1] A. Wright, "Ready for a web OS?" *Commun. ACM*, vol. 52, no. 12, pp. 16–17, 2009.

[2] A. Weiss, "WebOS: Say goodbye to desktop applications," *netWorker*, vol. 9, no. 4, pp. 18–26, 2005.

[3] D. Benslimane, S. Dustdar, and A. P. Sheth, "Services mashups: The new generation of web applications," *Internet Computing*, vol. 12, no. 5, pp. 13–15, 2008.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*. RFC Editor, 1999.

[5] S. Ihm and V. S. Pai, "Towards understanding modern web traffic," in *Internet Measurement Conference*, 2011, pp. 295–312.

[6] S. Souders, *High Performance Web Sites -Essential Knowledge for Front-End Engineers-*. O'reilly, 2007.

[7] G. Frederick and R. Lal, *Optimizing Mobile Markup*. Apress, 2009, pp. 213–238.

[8] Z. Liu, Y. Saifullah, M. Greis, and S. Sreemanthula, "HTTP compression techniques," in *Wireless Communications and Networking Conference*, vol. 4, Mar. 2005, pp. 2495–2500.

[9] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley, "Network performance effects of HTTP/1.1, CSS1, and PNG," in *Conf. Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1997, pp. 155–166.

[10] G. A. Kildall, "A unified approach to global program optimization," in *Symp. Principles of Programming Languages*, 1973, pp. 194–206.

[11] J. Lecomte, "YUI compressor," http://yui.github.io/yuicompressor/, [Online; accessed 01-Dec-2014].

[12] Google Inc., "Closure compiler," https://developers.google.com/closure/compiler/, [Online; accessed 01-Decv-2014].

[13] D. Edwards, "A javascript compressor. version 3.0," http://dean.edwards.name/packer/, [Online; accessed 01-Dec-2014].

[14] M. Burtscher, B. Livshits, G. Sinha, and B. G. Zorn, "JSZap: Compressing javascript code," in *USENIX Conf. Web Application Development*, 2010, pp. 39–50.

[15] A. B. King, *Website Optimization*. O'Reilly, 2008.

[16] F. Qian, J. Huang, J. Erman, Z. M. Mao, S. Sen, and O. Spatscheck, "How to reduce smartphone traffic volume by 30%?" in *Int'l Conf. Passive and Active Measurement*, 2013, pp. 42–52.

[17] Google Inc., "Data compression proxy," https://developer.chrome.com/multidevice/data-compression, [Online; accessed 01-Dec-2014].