2014 年度

修士論文

JavaScript 最適化と HTTP 圧縮による Web 通信量削減の実証的研究

神戸大学大学院システム情報学研究科

計算科学専攻

132X214X 坂元 康好

指導教員	中村 匡秀 准教授
審査教員	主査 鳩野 逸生 教授
	副査 上原 邦昭 教授
	副査 羅 志偉 教授
	副査 中村 匡秀 准教授

2015 年 2 月 9 日

Evaluating The Effect of Script Minification and HTTP Compression for Reducing Web Traffic

Yasutaka SAKAMOTO

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 optimization 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 are summarized as follows.

(1) 87% of top 500 websites have a certain amount of reduction potential by script optimization.

(2) 92% of top 500 websites have already been configured to support HTTP compression, whereas most non-popular websites such as Japanese government universities have not yet.

(3) 39% of the total size of JavaScript files used on the top 500 websites can be potentially reduced by applying script optimization.

(4) HTTP compression is saving over 50% of the current data traffic size of total JavaScript files. If every website was configured to use HTTP com-

pression, we can save a further 5% to 20% of JavaScript traffic.

JavaScript 最適化と HTTP 圧縮による Web 通信量削減の実証的研究

坂元 康好

要旨

コードオンデマンドは生スクリプトファイルを動的にダウンロードし，クライアントで実行するアーキテクチャスタイルの一つである．この形式は生スクリプトファイルが必ずしも事前にコンパイルまたは最適化されているとは限らないため，ネットワークのトラヒックに関する問題が発生する．可読性を確保するためのインデントや改行，コメントのような書式のルールは必ずしも実行には不要である．データ転送量の節約と処理時間の観点からそのようなコードは削除される必要がある．

本稿では 2 つのサイズ削減の手段（スクリプト最適化，HTTP 圧縮）に着目し，JavaScript のサイズ削減による通信量削減の潜在能力を明らかにする．

本稿の研究課題は以下の 2 つである．

RQ1: どのくらいの *Web* サイトにサイズ削減の余地があるか．

RQ2: 実際に *Web* 上にある *JavaScript* のサイズをどの程度削減できるか．

結果をまとめると以下のようになる．

（*1*）アクセスランキングトップ *500* の *Web* サイトの内，*87%* がスクリプト最適化で *JavaScript* のファイルサイズを削減できる余地がある．

（*2*）トップ *500* の *Web* サイトの内，*92%* がすでに *HTTP* 圧縮の設定がされている．その一方で，日本の国立大学のような頻繁にアクセスされない *Web* サイトでは，*85%* の *Web* サイトで *HTTP* 圧縮の設定がされていない．

（*3*）トップ *500* の *Web* サイトで使われている *JavaScript* の総サイズの *39%* がスクリプト最適化を行うことで *JavaScript* のファイルサイズを削減することができる．

（*4*）*HTTP* 圧縮により，現在 *JavaScript* のファイルにかかる現在のデータトラヒック量を *50%* 以上削減することができている．さらに全ての *Web* サイ

トで *HTTP* 圧縮の設定を行うと，*5%* から *20%* の *JavaScript* のトラヒック量を削減できる．

目次

図目次

表目次

第 1 章

はじめに

HTML5 を始めとする様々な Web 技術の登場により，Web ブラウザの OS 化が進んでいる [1][2]．この考えを一般にブラウザ OS と呼ぶ．これは，従来ネイティブアプリケーションの形で提供されていたあらゆるソフトウェアを，ブラウザ一つで実行するという考えである．ブラウザ OS の世界では，ブラウザ自体はソフトウェアを支える最低限の機能だけを提供し，利用者の求める具体的なアプリケーションはインターネットを経由して提供・利用される．実際に現在では，メールやチャット，SNS などのテキストを主体とする比較的単純なアプリケーションだけでなく，GPU を用いた 3 次元のゲームやシミュレーションのほか，エディタやコンパイル・デバッグ環境を備えた統合開発環境など，様々なアプリケーションをブラウザ一つで利用することができる．

このブラウザ（あるいは Web）を中心とするコンピューティング環境では，JavaScript が重要な責務を持つ．元来，JavaScript はクライアント側で実行できる手軽なスクリプト言語として登場し，動的な HTML 文章を実現するために利用されていた．2005 年には Google Maps が登場し，JavaScript を用いた非同期通信（Ajax）とインタラクティブな UI の構築方法が広く知られるようになる．これを契機として JavaScript の持つ可能性が見直されるようになり，その機能は現在でも拡張され続けている．現在では，サーバサイド機能の呼び出し処理や，クライアントとサーバ間のリアルタイム通信，ユーザによる UI 操作の制御，ブラウザから発効されるイベント処理などの機能を持っており，Web に欠かせない技術となっている．

JavaScript の一つの特色として，コードオンデマンドというスタイルが挙げ

られる（クライアントスクリプティングとも呼ばれる）．これは REST と呼ばれるソフトウェアのアーキテクチャスタイル内で定義されたスタイルの一つである [3]．コードオンデマンドに従ったアーキテクチャでは，クライアントはソースコード自体をサーバからダウンロードし実行する．事前にソースコードをコンパイルしておき，実行可能なバイナリファイルを配布するアーキテクチャとは対照的な方法といえる．コードオンデマンドの利点は，サーバ側のプログラムに修正を加えることなく柔軟にクライアントの機能を拡張・修正できる点にある．また，ソースコードを直接配布するという特性上，開発者によるアップデートの制御が容易という利点も持つ．

しかしながら，コードオンデマンドはネットワークの通信効率にデメリットを持つ．サーバがソースコードそのものを提供するという特性上，コンパイラによる最適化やサイズ縮小化が保証されない．そのため，ソースコード中の無駄な記述が含まれる可能性が常に発生する．例えば，可読性を確保するための書式ルール（インデントや改行，コメントなど）は，その実行にはそもそも不要である．また，プログラム上の論理的な無駄な記述（到達不能コードや使用されない変数宣言，デバッグ用コードなど）も不要なコードの一種である．通信効率化のためには，これらの無駄な記述を事前に削除した上でサーバ上で公開されるべきである．

Web サーバソフトウェアにも通信効率を低下させる要因がある．HTML や CSS，JavaScript 等の Web コンテンツはテキストファイルであるため，高い圧縮効果が見込める [4]．特にハフマン符号化などの可逆圧縮は，通信効率を向上させる有効な手法である．しかしながら，HTTP/1.1 ではクライアントとサーバ間での可逆圧縮通信を規定しているにも関わらず，ほとんどの Web サーバソフトウェアは標準でその機能を無効化している．サーバでの圧縮処理とクライアントでの展開処理が発生するため，（わずかながら）両者の負荷が増大するためである．効率的な通信を実現するためには，サーバ管理者が意図して圧縮通信を有効にする必要がある．

インターネット全体での通信効率化のためには，Web 全体に存在するあらゆる JavaScript が事前に最適化されていることが望ましい．現在，光回線等の高速な通信インフラが普及しつつあるが，インターネット通信の効率化には

高い需要があると我々は考える．未だ高速な通信インフラが整備されていない国も数多く存在するほか，空港やカフェ等での混雑した低速 Wi-Fi 環境もまだまだ珍しくない．モバイル端末での通信インフラの多くは従量課金制であり，無駄なデータ転送は可能な限り避けたいという声も多い．さらに，今後の Web の普及や発展を考えれば，Web コンテンツサイズの肥大化はもはや避けられないといえる．

通信効率化の観点では，JavaScript ではなくマルチメディアコンテンツ（画像や動画，音声など）を改善するという方法も考えられるが，その効果は期待しにくい．基本的に JavaScript などのテキストベースコンテンツと比べて，マルチメディアコンテンツは単体でのファイルサイズが極めて大きい．Ihm と Pai らの調査 [5] によると，インターネット通信量全体の 50% を占める．しかしながら，マルチメディアコンテンツに対する圧縮処理の適用はすでに広く知れ渡っており，もはや情報リテラシの一種とも見なすことができる．一方で，JavaScript に対する事前の最適化処理は，まだ共通認識としての浸透には至っていない．単体でのファイルサイズが小さいため，軽視されているともいえる．

本研究の目的は，Web 全体における，JavaScript サイズ削減による通信量削減の潜在能力を明らかにすることである．もし，Web 上に点在する JavaScript 全体での高いサイズ削減効果が確認できれば，そのサイズ削減手段は，Web エンジニアの共通認識，あるいはリテラシとして普及されるべきといえる．本研究ではサイズ削減の手段として，スクリプト最適化 [6][7] と HTTP 圧縮 [8][9] の 2 つに着目する．スクリプト最適化は，実行に不要なコード記述の削減やプログラムロジックの組み替え処理を行うことで，サイズという観点でソースコードを最適化し縮小化する方法である．HTTP 圧縮は HTTP/1.1 で規定されている機能であり，転送する HTTP パケットをサーバ側で圧縮し，クライアント側で展開する方法である．

調査方法としては，まずアクセス数の多いトップ 500Web サイトを題材として，JavaScript ファイルに対する HTTP 圧縮の適用状況を調べる．さらに，これらの Web サイトに含まれている JavaScript コードを回収し，実際にサイズ最適化処理を施すことで，スクリプト最適化の潜在的な効果を調べる．

本研究の研究課題は以下の 2 つである．

RQ1: どのくらいの *Web* サイトにサイズ削減の余地があるか．

サイズを削減できる可能性のある Web サイトの割合を調査する．この割合は，JavaScript サイズの削減において一つの潜在能力とみなすことができる．

RQ2: 実際に *Web* 上にある *JavaScript* のサイズをどの程度削減できるか．

スクリプト最適化と HTTP 圧縮を用いることでどの程度の削減率が得られるかを調査する．サイズ削減の余地がある JavaScript が多く残っている時，より効率的にネットワークを利用できる可能性がある．

これらの二つの研究課題について調査する．

第 2 章

JavaScript 最適化と HTTP 圧縮による Web 通信量の削減

2.1 諸定義

初めに，以下の 2 つの言葉を定義する．

最適化: スクリプトが最適化されているかどうかを判定することは，最適化処理が多様なため容易ではない．本稿では，スクリプトに 1 カ所以上インデントまたはコメントが含まれていた場合，最適化されていないと定義する．この処理は手作業で行う．

最適化率と削減率: JavaScript の最適化率と削減率は本研究では重要な指標となる．最適化率はスクリプト最適化や HTTP 圧縮を使用することでどの程度 JavaScript のファイルサイズを小さくできたかを表している．また，削減率は JavaScript のファイルサイズをどの程度削減できたかを表している．本稿では，最適化率と削減率を以下のように求める．

$$OptimizationRate(\%) = \frac{ProcessedSize}{UnprocessedSize} \times 100 \tag{2.1}$$

$$ReductionRate(\%) = 100 - OptimizationRate \tag{2.2}$$

2.2 JavaScript の最適化 ($Script_{opt}$)

スクリプト最適化はスクリプトの処理に全く修正を加えずにサイズを削減するという観点で最適化・縮小化する技術である．本稿では，$Script_{opt}$ と呼ぶ

こととする．$Script_{opt}$ の具体例をあげる．次のコードは，0 から 10 までの整数の和を求める JavaScript のコードである．このスクリプトのサイズは 68 バイトである．

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

コードを読みやすくするためにインデントが追加されていたり，コメントが書かれているが，$Script_{opt}$ はそのような書式のルールを取り除くことをの基本処理としている．

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

上記のコードのように，全ての改行や不要なスペースが取り除かれた．この処理によるスクリプトのサイズ削減率はおよそ 24% である．

最も強力な最適化処理はプログラムロジックを組み替える処理である．この最適化はプログラム最適化の一種とみなすことができる [10]．例えば，一度も使われない変数や到達不能コードは実行には不要であり，それらは除去してもプログラムの振る舞いは変わらない．さらに，“true”，“false”，“undefined” はそれぞれ，“!0”，“!1”，“void 0” に短縮できる．

```
alert(55);
```

上記のコードはこの処理によって最適化が行われている．結果として，最適化後のファイルサイズは 10 バイトになり，85% の JavaScript を削減できた．

また，様々な種類の最適化ツールが Web 上で公開されている．ツールにより最適化の処理内容，処理時間，処理結果は異なる．本研究では，ツールの特徴を確認するために JavaScript 最適化ツールを 3 種類選択し，処理結果を比較する．

2.3　HTTP 圧縮 ($HTTP_{comp}$)

HTTP 圧縮は HTTP パケットを圧縮することで Web コンテンツの通信トラヒックを削減する効果的な機能であり，HTTP/1.1 [3] ではオプションで設

定ができる [9]．HTTP 圧縮を設定すると，Web コンテンツはクライアント側にデータが転送される前にサーバ側で圧縮される．一般に，gzip 形式が圧縮のアルゴリズムとして使われている．gzip 形式とは，ハフマン符号を用いた Lempel-Ziv (LZ77) のアルゴリズムを採用している．本稿では，$HTTP_{comp}$ と呼ぶことにする．

$HTTP_{comp}$ を設定した場合のネゴシエーションの履歴を一例として以下に示す．“GET” はページの取得要求，“Host” はリクエスト先のサーバ名，“Accept-Encoding” は Web ブラウザが受信可能なエンコーディング形式を表すフィールドである．初めに，クライアントは $HTTP_{comp}$ を有効にするため，“Accept-Encoding” のフィールドを含んだリクエストメッセージをサーバに送信する．

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

次に，サーバは以下のレスポンスメッセージを送信する．ここで “200 OK” は HTTP のステータスコードであり，クライアントが送信したリクエストが成功したことを表している．また，“Content-Type” は取得したコンテンツの種類，“Content-Encoding” はクライアントに送信するコンテンツのエンコーディング形式を表すフィールドである．

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

この場合, レスポンスとして送られた HTML コンテンツ（http://example.com/index.html）は gzip で圧縮した状態で転送されている．クライアントは HTML コンテンツを解析する前にレスポンスを復元する必要がある．$HTTP_{comp}$ は，Apache HTTP Server, Internet Information Services，Nginx などの Web サーバによりサポートされているが，標準設定では利用できない．これは，データを圧縮，復元時にサーバやクライアントにわずかながら負荷がかかるためであ

る．このことから HTTP 圧縮を設定していない Web サイトが存在するため，HTTP 圧縮を用いると JavaScript を転送するために必要なデータトラヒックを削減する余地がある．

HTTP ヘッダは上記以外にも数多く送受信されている．本実験で取得した主な HTTP ヘッダを表 2.1 に示す．

2.4　予備実験

$Script_{opt}$ と $HTTP_{comp}$ の削減効果を確かめるために，予備実験を行った．実験対象として，よく利用されている JavaScript ライブラリである，jQuery (ver. 2.1.1)， prototype.js (ver. 1.7.2)， backbone.js (ver.1.1.2) とそれらをすでに最適化している状態の JavaScript（jquery.min.js，prototype.min.js，backbone.min.js ）を用いた．最適化された状態の JavaScript は Web サイトで公式に提供されている．以後，最適化処理が行われていない状態の JavaScript（jquery.js，prototype.js，backbone.js ）を raw JS，すでに最適化が行われている JavaScript（jquery.min.js，prototype.min.js，backbone.min.js ）を min JS と呼ぶ．2.2 節で述べたように，様々な種類の最適化ツールが公開されている．その中から，今回は YUI compressor を使用した．YUI compressor はインデント，コメント，改行の削除やローカル変数の変数名を短縮をすることでスクリプトを最適化している．

予備実験の結果を表 2.2 に示す．“unprocess” は最適化処理を行っていない状態，“$Script_{opt}$”，“$HTTP_{comp}$” はそれぞれスクリプト最適化と HTTP 圧縮を行った状態，“Both ” は両方の処理を行った状態でのファイルサイズの結果である．括弧内の割合は各ライブラリの “unprocess” のファイルサイズを基準とした最適化率を表している．raw JS は，$Script_{opt}$ を行うことにより，ライブラリのファイルサイズはおよそ 30% から 50% に最適化されている．すなわち，50% から 70% ファイルサイズを削減できたことになる．一方，min JS は $Script_{opt}$ を行うとファイルサイズが増加してしまった．これは，YUI の処理の一つとして，最適化後のプログラムを正常に動作させるために括弧が自動的に補完されたためであると考えられる．よって，既に最適化されている JavaScript に $Script_{opt}$ による削減の余地はない．次に，$HTTP_{comp}$ は全ての

Table 2.1: List of HTTP headers

(a) Request Header

Header Field	Description
accept	Web ブラウザが受信可能な形式
accept-encoding	Web ブラウザが受信可能なエンコーディング形式
connection	サーバ間との持続的接続に関する情報
cookie	クッキー
host	リクエスト先のサーバ名
user-agent	Web ブラウザの情報

(b) Response Header

Header Field	Description
accept-ranges	使用可能な単位
cache-control	キャッシュに関する命令
content-encoding	コンテンツのエンコーディング形式
content-length	コンテンツのサイズ
content-type	コンテンツの種類
date	応答した時刻
etag	リソースに関する情報．キャッシュの有効性を確認する
expires	エンティティの有効期限
last-modified	最終更新時刻
location	Weg ブラウザが受信可能な形式
server	サーバ情報
transfer-encoding	転送に使用するエンコーディング形式

JavaScript で約 70% の高い削減率が得られた．これはハフマン符号はテキストベースのファイルに効果的な結果を与えるため，最適化されていることは関係なく高い削減率が得られた．

2 つを組み合わせると，約 90% の削減率が得られた．この理由は，$Script_{opt}$ の処理結果はテキストベースの JavaScript であり，$HTTP_{comp}$ はテキストベースのファイルに効果的であるためである．従って，$Script_{opt}$ と $HTTP_{comp}$ はそれぞれ削減率に寄与すると考える．

2.5　本研究のスコープ

本研究の研究課題は以下の二つである．

RQ1: どのくらいの *Web* サイトにサイズ削減の余地があるか．

まず初めに，Web 上の実際の削減率を調査する前に，Web サイトに含まれる JavaScript ファイルがサイズ削減が可能であるかを調査する．サイズ削減可能である Web サイトの割合は JavaScript のサイズ削減する余地があることを示す指標の一つとみなすことができる．調査内容として，JavaScript ファイルに対する $Script_{opt}$ と $HTTP_{comp}$ の適用状況を調査する．

RQ2: 実際に *Web* 上にある *JavaScript* のサイズをどの程度削減できるか．

この課題は本研究で最も重要になる．実際に Web サイトにアクセスし，$Script_{opt}$ と $HTTP_{comp}$ を使用した場合，どの程度 JavaScript のファイルサイズを削減できるかを実際に調査する．サイズ削減の余地がである JavaScript が未だ多く存在している場合，混雑した場所や Wi-Fi の通信速度が遅い場所で効率的に帯域幅を利用できる余地があるといえる．この課題を調査するため，Web サイトで使われている全ての JavaScript コードを回収し，その JavaScript に $Script_{opt}$ と $HTTP_{comp}$ を適用し，結果を比較する．

Table 2.2: Reduction effects of $Script_{opt}$ and $HTTP_{comp}$

Subject lib.	unprocessed	$Script_{opt}$	$HTTP_{comp}$	Both
jQuery.js	241.6KB	128.2KB	72.9KB	37.0KB
	—	(53%)	(30%)	(15%)
jQuery.min.js	82.3KB	82.7KB	28.8KB	28.9KB
	—	(100%)	(34%)	(35%)
prototype.js	193.1KB	102.3KB	45.3KB	33.0KB
	—	(53%)	(23%)	(17%)
prototype.min.js	95.1KB	96.4KB	29.9KB	31.4KB
	—	(101%)	(31%)	(33%)
backbone.js	59.6KB	19.7KB	17.3KB	6.9KB
	—	(33%)	(29%)	(12%)
backbone.min.js	19.5KB	19.8KB	6.5KB	6.9KB
	—	(101%)	(33%)	(35%)

第 3 章

Web 通信量削減の実証実験の設計

3.1 調査対象

調査対象として，Alexa 社で公開されている世界アクセスランキングトップ 500[*1]のリストを用いた．Web サイトのアクセスランキングのリストは Web 全体で実証的に調査する上で適している．特に，人気があり，頻繁にアクセスされている Web サイトに削減の余地が多くあるならば，ネットワークのトラヒック量を削減する上でとても効果的である．以後，このリストをトップ 500 と呼ぶこととする．ここで，Alexa 社の世界アクセスランキングのデータはユーザの Web ブラウザにインストールされている “Alexa Toolbar” を通して集計されている．よって，結果に偏りが生じていることに注意する必要がある．集計結果の偏りは Alexa Toolbar を使用するユーザに影響する．

さらに，日本の国立大学の Web サイトのリストも用いた．日本には国立大学が 86 校ある．現在の World Wide Web には莫大な数の Web サイトが含まれており，さらに増加し続けている．研究課題を解決するためには，人気のある Web サイトだけでなく，あまりアクセスされない Web サイトも調査する必要がある．従って，その中のドメインの一つとして日本の国立大学を取り上げた．以後，大学と呼ぶことにする．

ここで，ルートディレクトリに存在する Web ページは同じ Web サイトに含まれる他の Web コンテンツに強く反映されている傾向があるため，特定の Web サイトのルートドメイン（すなわち，ルートに存在するインデックスページ）のみを本実験では対象としていることに注意が必要である．

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

3.2　実験の流れ

RQ1，RQ2 に対する実験の流れを図 3.1 に示す．

1. *Fetching to root domain:* 特定のドメイン名（図 3.1 の場合，“example.com”）に対し，ルートディレクトリへ HTTP の GET リクエストを送信し，ルートにあるインデックスページを取得する．取得先のページにリダイレクトが指定されている場合，リダイレクト先の Web サイトのルートのインデックスページを取得する．また，取得先の Web サイトが見つからなかった場合はその Web サイトを実験から除外する．これを，トップ 500，大学のリストに含まれる全ての Web サイトで行う．
2. *JavaScript extraction:* スクリプトを抽出するためにレスポンスの本文を DOM パーサで解析する．HTML ドキュメントに直接書き込まれているスクリプト（図 3.1 の場合，“internal JS”）を内部 JavaScript と呼ぶ．内部 JavaScript はドキュメントの`<script>`タグ内に記述されているので，`<script>`タグ毎に抽出し，保存する．
3. *Fetching external JavaScript:* DOM パーサで解析した時に，HTML ドキュメントに外部 JavaScript（図 3.1 の場合，“external JS”）を含むものもある．外部 JavaScript とは，ドキュメントに直接書き込まれている JavaScript ではなく，外部から参照されている JavaScript のことである．外部 JavaScript はクライアント側で動的に呼び出されている．この場合，ドキュメントには`<script src="ext.js">`のように参照先が記述されている．それらを参照先からそれぞれ呼び出し，保存する．
4. *Storing HTTP header to analyze* $HTTP_{comp}$*:* Step.3 で外部 JavaScript と同時に取得した HTTP のレスポンスヘッダの情報をデータベースに保存する．このヘッダには外部 JavaScript を呼び出す時に，HTTP 圧縮の設定や転送時のファイルサイズに関する情報が含まれている．HTTP 圧縮の設定に関しては RQ1，転送時のファイルサイズに関しては RQ2 で必要な情報となる．

5. *Merging all JavaScript:* 内部 JavaScript と外部 JavaScript を実行される順番を保持したまま，1 つの JavaScript（図 3.1 の場合，all.JS）にまとめる．
6. *Applying JavaScript optimization:* all.JS を 3 種類の JavaScript 最適化ツールを用いて最適化する．JavaScript 最適化ツールについては次節で詳細に説明する．
7. *Storing optimization results to analyze* $Script_{opt}$*:* 削減率や最適化にかかる処理時間などの $Script_{opt}$ の実験結果を保存する．この結果は $Script_{opt}$ の分析に使用する．

3.3　JavaScript 最適化ツール

Web 上に公開されている様々な JavaScript 最適化ツールから次の 3 種類のツールを選択した．YUI Compressor は適応性が高いが削減率が低いものである．対照的に，Closure Compiler は削減率は高いが適応性は低いものである．Dean Edwards' JavaScript packer は難読化ツールである．各ツールについて詳細に説明する．

YUI Compressor[11] (YUI) は一般的によく知られている JavaScript 最適化ツールである．YUI はフォーマットの最適化（スペース，インデント，改行の削除）とプログラムロジックの最適化を処理の内容としている．

最適化のオプションとして，以下のものがある．

- **Minify only, do not obfuscate**（縮小化のみ行い，難読化しない）
  ローカル変数の変数名を変更せず，そのままの変数名で使用する．
- **Preserve all semicolons**（全てのセミコロンを保存しておく）
  不要なセミコロンを削除せず，そのまま残しておく．
- **Disable all micro optimizations**（最適化を無効にする）
  記述方法の変更によるソースコードの最適化をせずに，そのままの状態にしておく．

これらのオプションは最適化を行う時に選ぶことができる．選択するオプショ

ンが多いほど，コードの可読性は高まるが，最適化によるサイズ削減効果は小さくなる．refscrmin 節で述べた，全ての改行と不要な空白が削除された結果のスクリプトは YUI のオプションをどれも選択しなかった時の結果である．

このツールの利点として，他の最適化ツールと比較した場合，削減後のファイルへの影響が小さいことが挙げられる．本稿ではこのツールを $Script_{opt}$ の基準とする．

図 3.2 のソースコードは，リストに含まれる数を昇順で並び替え，その結果を表示する JavaScript のプログラムである．YUI を用いてこの JavaScript プログラムを最適化を行った結果を図 3.3 に示す．オプションは選択していない．最適化後のソースコードはスペース，インデント，改行が削除され，変数名が一部短縮化された．

Closure Compiler[12] (CC) は Google によって提供されている最適化と縮小化を行うツールである．CC は YUI とは対照的にスクリプトのプログラムロジックを積極的に最適化する．

最適化レベルとして，以下がある．

- **WHITESPACE_ONLY**
  コメント，改行，スペースの削除を行う．
- **SIMPLE_OPTIMIZATIONS**
  “WHITESPACE_ONLY” の処理に加えて，関数内のローカル変数と引数名を短縮する．
- **ADBANCED_OPTIMAZATIONS**
  “SIMPLE_OPTIMIZATIONS” の処理に加えて，全てのシンボル名の変換，一度も使用していない変数・関数を削除する．また，関数のインライン化を行う．関数のインライン化とは，関数呼び出しを関数の内容そのものに置き換える処理である．

最適化のレベルは，実行時に一つを選ばなければならない．2.2 節で述べた，最も強力なサイズ削減であるプログラムロジックの組み替えを行った結果は CC の “ADBANCED_OPTIMAZATIONS” によって処理されたものである．しかし，CC は最大限の最適化コードを引き出すために最適化を行うスクリプ

トに記述方法に関していくつかの制約を課している．よって，CC はサイズ削減効果は高いが適応性が低いツールである．

図 3.2 のソースコードを CC を用いて最適化した結果を，図 3.4 に示す．最適化のレベルは “ADBANCED_OPTIMAZATIONS” とした．最適化後のソースコードはスペース，インデント，改行の削除だけでなく，変数名も YUI と比較するに短縮されており，積極的に最適化を行っていることがわかる．

Dean Edwards’ JavaScript packer[13] (packer) は JavaScript 難読化ツールである．一般に，難読化は可読性を低くするためにスクリプトのプログラムロジックを大幅に変える処理である．

難読化のレベルして，以下がある．

- **None**
  難読化は行わず，最適化のみ行う．最適化の処理として，スペース，コメントの削除と変数名の短縮を行う．
- **Numeric (Base 10)**
  ソースコードを数値列に変換し，難読化する．
- **Normal (Base 62)**
  ソースコードを英数字列に変換し，難読化する．
- **High ASCII (Base 95)**
  ソースコードを英数字列に変換し，難読化する．文字コードは High ASCII を使用する．

これらは実行時に一つを選ばなければならない．

“None” 以外のレベルで難読化されたスクリプトは常に次のように始まる．

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

難読化と最適化は処理は似ているが，処理の目的や原則において全く異なるものである．しかし，packer の難読化は，スクリプト最適化と同様の結果が得られる．よって，実験で用いる最適化ツールの一つにこのツールを選んだ．

図 3.2 のソースコードを packer を用いて最適化した結果を，図 3.5 に示す．最適化のレベルは “Normal (Base 62)” を用いた．最適化後のソースコードは，難読化され，ソートを行うプログラムであるかは見た目からでは判断できなく

なった．今回ソートのプログラムが単純であるため，最適化後の JavaScript のサイズが最適化前よりも大きくなってしまった．しかし，たいていの場合，最適化後のサイズの方が小さくなり，効果的な削減率が得られる．

前述の通り，3 種類のツールには全て最適化や難読化のためのアルゴリズムを選択するためのオプションがある．本実験で用いるツールのオプションを脚注に示す．[*2]

3.4　実装

Web サイトからルートにあるインデックスページを取得し，JavaScript を抽出するために Ruby を用いた．その後，JavaScript を 3 種類の最適化ツールを用いて最適化するために Perl を利用した．また，$Script_{opt}$ で取得したルートインデックページ，HTTP ヘッダ情報，インデックスページから抽出した JavaScript，その JavaScript に関する情報を保存するために mongoDB を利用した．開発工数は約 1 人月要した．プログラム総行数は，Ruby は約 600 行，Perl は約 150 行である．

[*2] **YUI Compressor:** 標準設定を使用する．YUI の場合，この設定は最大の最適化効果が表れる．処理として，フォーマット形式の削除，ローカル変数の置き換えを行う．

Closure Compiler: “ADBANCED_OPTIMAZATIONS” を使用する．このオプションは CC の中で最も積極的に最適化を行う設定であり，論理的な振る舞いを考慮しつつ全体的にコードを最適化する．最適化を行うコードが記述に関する制約を守られていない場合，コンパイルエラーが発生する．

packer: “Normal (Base 62)” を選択する．コードは難読化の処理が行われ 62 種類の英数字列（a-z, A-Z，0-9）に最適化される．

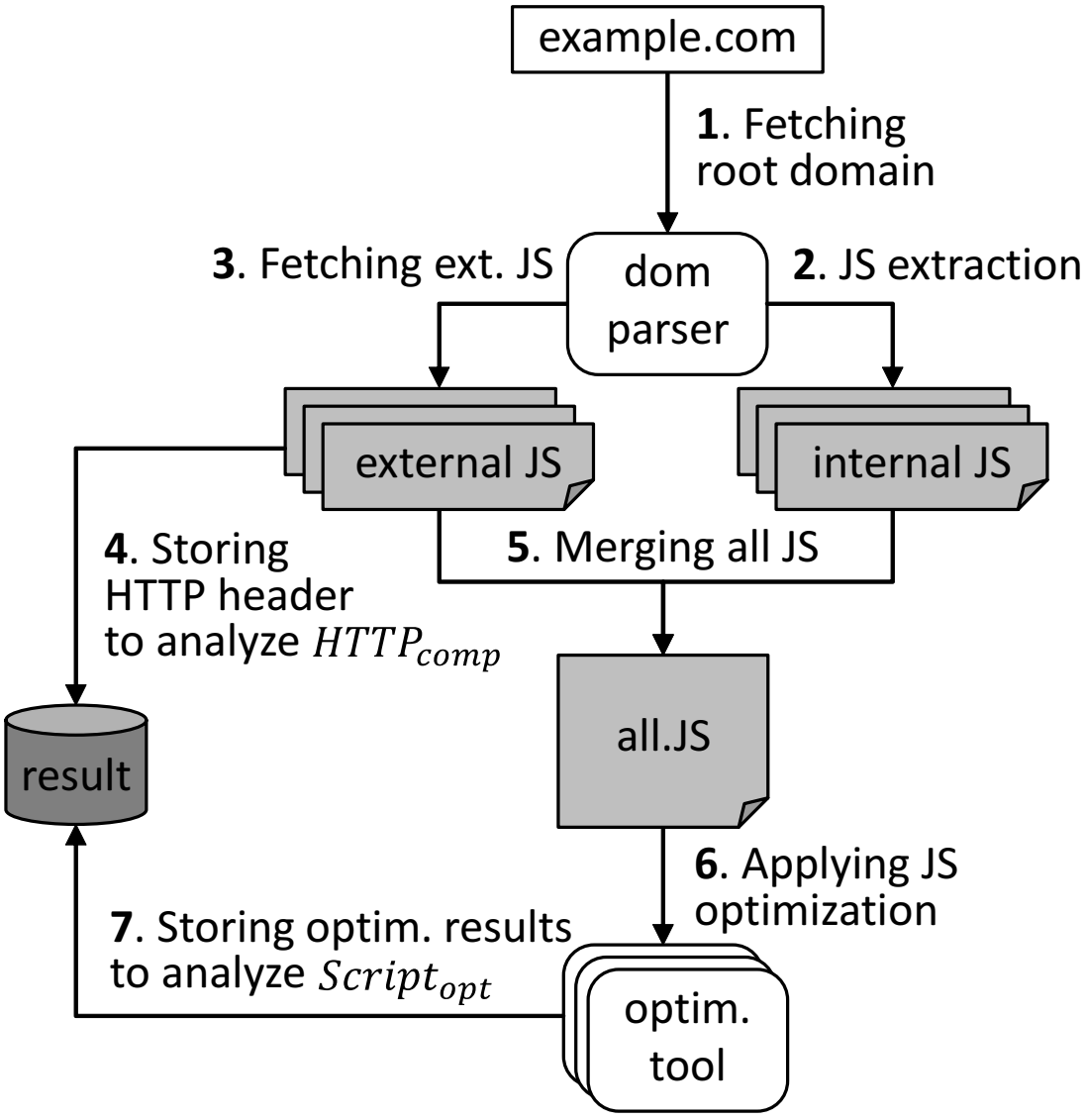


Fig. 3.1: Experimental Procedure

```
var list = [342, 221, 3, 546, 237, 849, 178, 763, 9];

function sort(list)
{
    var swap;
    do {
        swap = false;
        for (var i=0; i < list.length-1; i++) {
            if (list[i] > list[i+1]) {
                var tmp = list[i];
                list[i] = list[i+1];
                list[i+1] = tmp;
                swap = true;
            }
        }
    } while (swap);
}

sort(list);
console.log(list);
```

Fig. 3.2: Sort program

```
var list=[342,221,3,546,237,849,178,763,9];function
sort(c){var d;do{d=false;for(var b=0;b<c.length-1;b++)
{if(c[b]>c[b+1]){var a=c[b];c[b]=c[b+1];c[b+1]=a;d=true
}}}while(d)}sort(list);console.log(list);
```

Fig. 3.3: Sort program processed by YUI Compressor

```
var a=[342,221,3,546,237,849,178,763,9],b;do{b=!1;for(var
c=0;c<a.length-1;c++)if(a[c]>a[c+1]){var d=a[c];
a[c]=a[c+1];a[c+1]=d;b=!0}}while(b);console.log(a);
```

Fig. 3.4: Sort program processed by Closure Compiler

```
eval(function(p,a,c,k,e,r){e=function(c){return
c.toString(a)};if(!''.replace(/^/,String)){while(c--
)r[e(c)]=k[c]||e(c);k=[function(e){return r[e]}];
e=function(){return'¥¥w+'};c=1};while(c--)if(k[c])
p=p.replace(new RegExp('¥¥b'+e(c)+'¥¥b','g'),k[c]);return
p}('2 4=[l,j,3,d,g,e,n,7,9];8 5(a){2 b;f{b=6;h(2 i=0;
i<a.k-1;i++){m(a[i]>a[i+1]){2 c=a[i];a[i]=a[i+1];
a[i+1]=c;b=o}}}p(b)}5(4);q.r(4);',28,28,'||var||list|sort
|false|763|function|||||546|849|do|237|for||221|length|34
2|if|178|true|while|console|log'.split('|'),0,{}))
```

Fig. 3.5: Sort program processed by JavaScript packer

第 4 章

実証実験の結果と考察

4.1 実験結果の概要

本実験で用いる Web サイトのリストについてまとめたものを表 4.1 に示す．表 4.1 には，以下のものが示されている．

- #includedwebsites　リストに含まれる Web サイト数
- #notfoundwebsites　“Not Found” であった Web サイト数
- average#externalJSfiles　外部 JavaScript の平均ファイル数
- average#internalJSfiles　内部 JavaScript の平均ファイル数
- average#JSfiles　HTML に含まれる JavaScript の平均ファイル数
- averagesizeofJSfiles　JavaScript ファイルの平均サイズ
- totalsizeofJSfiles　JavaScript の総サイズ

トップ 500Web サイトの内，22 サイトで Web サイトが見つからなかったため本実験では除外した．トップ 500 に焦点を当てると，平均して JavaScript が 20 ファイル程度含まれていた．JavaScript の総サイズは 157MB である．このサイズは本実験において基準となる．この基準値を $Script_{opt}$ や $HTTP_{comp}$ でどの程度減らすことができるかが RQ2 の結果となる．

また，大学のリストでは，全ての大学で Web サイトが存在した．JavaScript の平均ファイル数は 12 ファイルであり，総サイズは 12MB であった．JavaScript の総サイズがトップ 500 と比較すると大幅に小さいのは，リストに含まれる Web サイト数が少ないためである．

次に，トップ 500 の Web サイトに含まれる，トップレベルドメインの割合を図 4.1 に示す．Web サイトのおよそ 70% は “com”，“net” ，“org” である．これにより，トップ 500 の Web サイトは商用の組織，ネットワークのインフラまたは組織に属していることが多いことがわかる．また，国別のコードトップレベルドメインはほとんど含まれていない．北東アジア（中国，日本，ロシア）とヨーロッパ（ドイツ，フランス，イギリス）が含まれている．これらの国は結果に強く影響を与えている可能性がある．

4.2　RQ1 の考察

初めに，$Script_{opt}$ の結果を説明する．トップ 500（図 4.2a）と大学（図 4.2b）の Web サイトにおける JavaScript 最適化の利用率を図 4.2 に示す．図中の “partially or not minified” は Web サイトにある一定量の削減の余地があることを表している．トップ 500 の Web サイトの 86.8% が $Script_{opt}$ を使用することでサイズ削減の余地があることが結果から読み取ることができる．日本の国立大学のようにトップ 500Web サイトと比べるとほとんどアクセスされない Web サイトでも削減の余地が大いにあることがわかった．

また，サイズ削減の余地がない Web サイトも存在した．原因として，トップ 500 では，

- “fully minified” : JavaScript が完全に最適化されている．（6.0%）
- “page not found” : Web サイトにアクセスできない．（4.6%）
- “no script” : Web サイトのルートにあるインデックスページに JavaScript が含まれていない（2.6%）

大学では，

- “page not found”（2.3%）
- “fully minified”（1.2%）

であった．JavaScript が含まれていない Web サイトや完全に最適化されているために削減の余地がない Web サイトも存在することがわかった．

次に，$HTTP_{comp}$ の結果を図 4.3 に示す．これはリスト内の Web サイトで

どの程度 HTTP 圧縮が設定されているかを割合で表している．図 4.3a より，トップ 500Web サイトでは，多くの Web サイトですでに $HTTP_{comp}$ を設定していることがわかる．一方，図 4.3b より，大学の Web サイトでは，$HTTP_{comp}$ を設定していない Web サイトが未だ数多く存在していることがわかる．

4.3　RQ2 の考察

$Script_{opt}$ の結果を図 4.4 ，表 4.2，トップ 10 の Web サイトの詳細結果を図 4.5 にそれぞれ示す．図 4.4 の “unprocessed” は Web サイトに含まれている最適化処理前の JavaScript の総サイズ，packer，CC，YUI はそれぞれの最適化ツールで最適化を行った後の JavaScript の総サイズを表している．注意すべきことは，“unprocessed” にはすでに最適化されたスクリプトと最適化・縮小化がされていないスクリプトのサイズがともに含まれていることである．図中に示されている値は削減率である．また，表 4.2 に書かれている “Reduction JavaScript Size” は，最適化処理による JavaScript の削減サイズ，“Average Processing Time” は最適化処理にかかる時間の平均，“Successfully Compiled Rate” は最適化処理の成功率である．packer はスクリプトを文字列とみなすことによりハフマン符号のアルゴリズムに基づいて最適化を行う．そのため，他の最適化ツールでは一度最適化されたスクリプトに対しては効果が小さいが，packer はそのようなスクリプトであっても高い削減率が得られる．この結果は図 4.5 でも見られる．

また，スクリプトを文字列とみなすことで書式ルールなどの文法エラーが発生しないため最適化処理の成功率はトップ 500，大学のリストともに 100% となった．一方，CC と YUI はスクリプトをプログラムと考えるため，文法上のミスがあると正常にコンパイルができない．さらに，CC は記述方法に独自の制約を課している．よって，その記述に従っていないスクリプトはコンパイルできないため，YUI と比べるとさらに成功率が低くなり，3 種類の最適化ツールの中で成功率が最も低くなった．

しかし，packer は他の 2 つツールと比べると処理に時間がかかることがわかった．

結果として，packer は最適化処理に時間はかかるが，約 39% の JavaScript

のサイズを削減することができた．

$HTTP_{comp}$ の結果を図 4.6 に示す．“unprocessed” は $HTTP_{comp}$ を適用しない場合に JavaScript ファイルの転送にかかる HTTP 通信のトラヒック量を表している．“fully-not gzipped” はリスト内の全ての Web サイトで $HTTP_{comp}$ を設定しない場合にかかる HTTP 通信のトラヒック量を表している．対照的に，“fully-gzipped” はリスト内の全 Web サイトで $HTTP_{comp}$ を設定した場合にかかる HTTP 通信のトラヒック量を表している．“lev.” は圧縮のレベルを表し，1 は最小，9 が最大，5 が標準である．レベルが上がるにつれて圧縮の効果が高くなる．図 4.6a の結果より，トップ 500 では現在 JavaScript ファイルの転送時に必要な通信トラヒック量は $HTTP_{comp}$ により半分以上削減されていることがわかる．さらに，全ての Web サイトで $HTTP_{comp}$ を使用すると，5% から 20% 通信トラヒック量を削減できると考えられる．また，図 4.6b の結果より，大学の Web サイトでは 59% から 65% の通信トラヒックを削減できることがわかった．従って，Web サイトには，$HTTP_{comp}$ によりトラヒック量を削減できる可能性が残っているといえる．

Table 4.1: 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

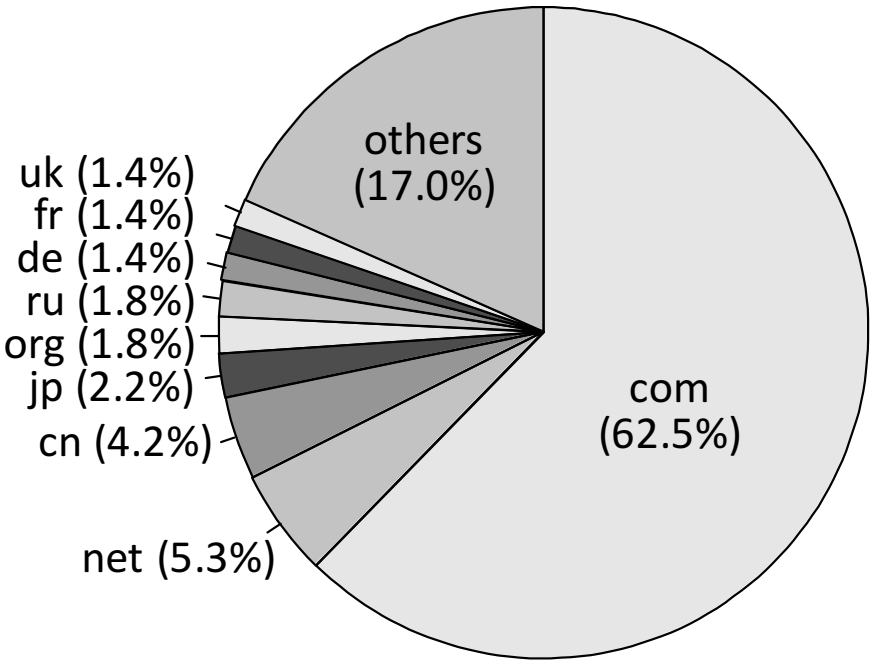
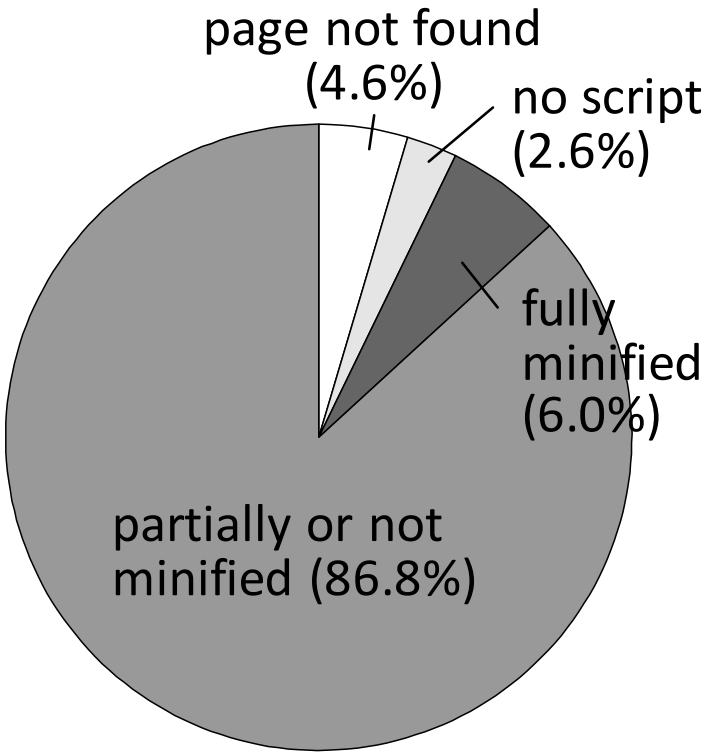
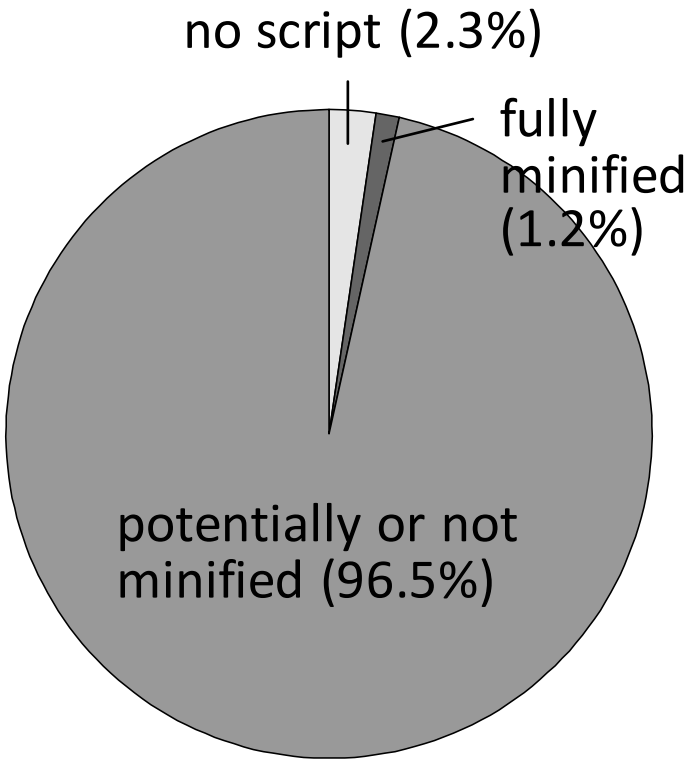


Fig. 4.1: Percentage of top-level domain of Alexa top 500 websites

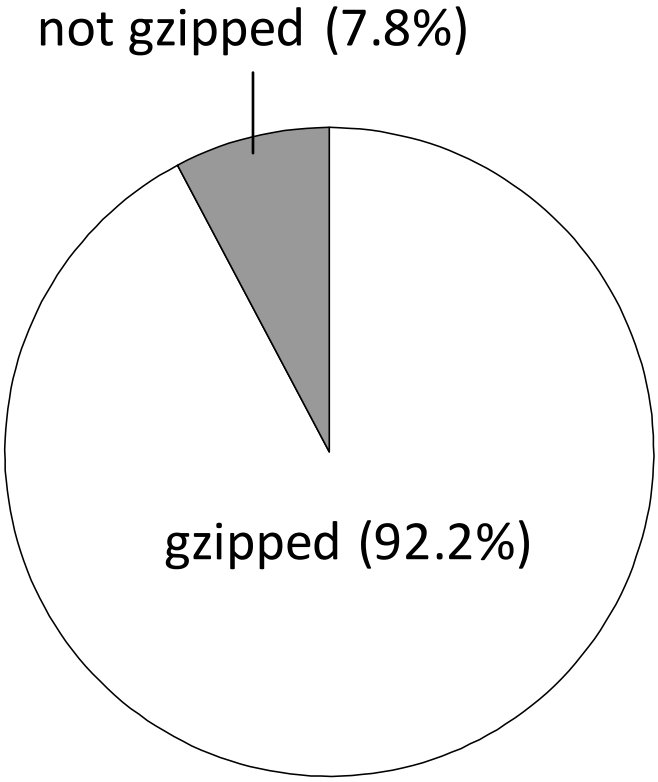


(a) Alexa top 500

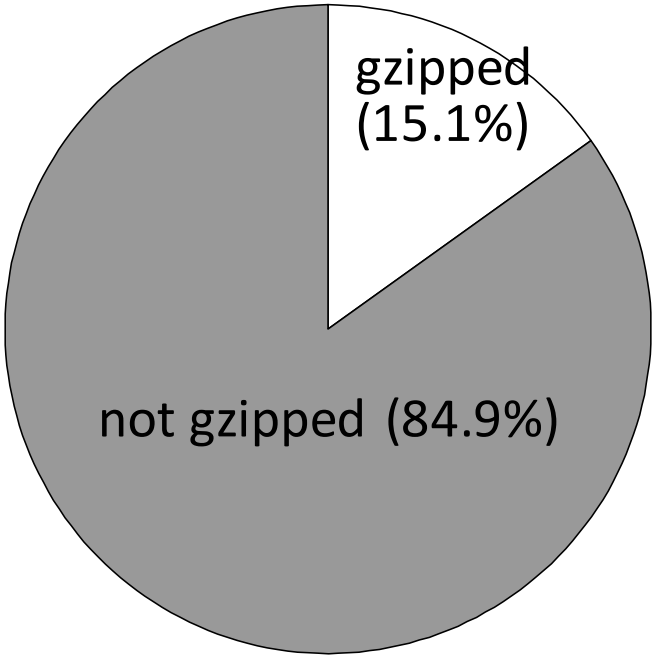


(b) Japanese gov. univ.

Fig. 4.2: Percentages of the usage of $Script_{opt}$

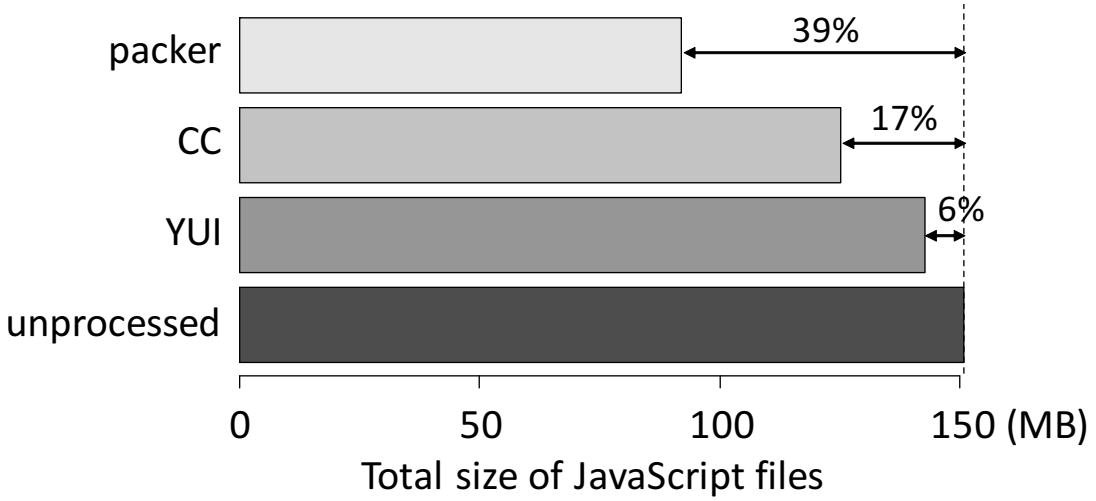


(a) Alexa top 500

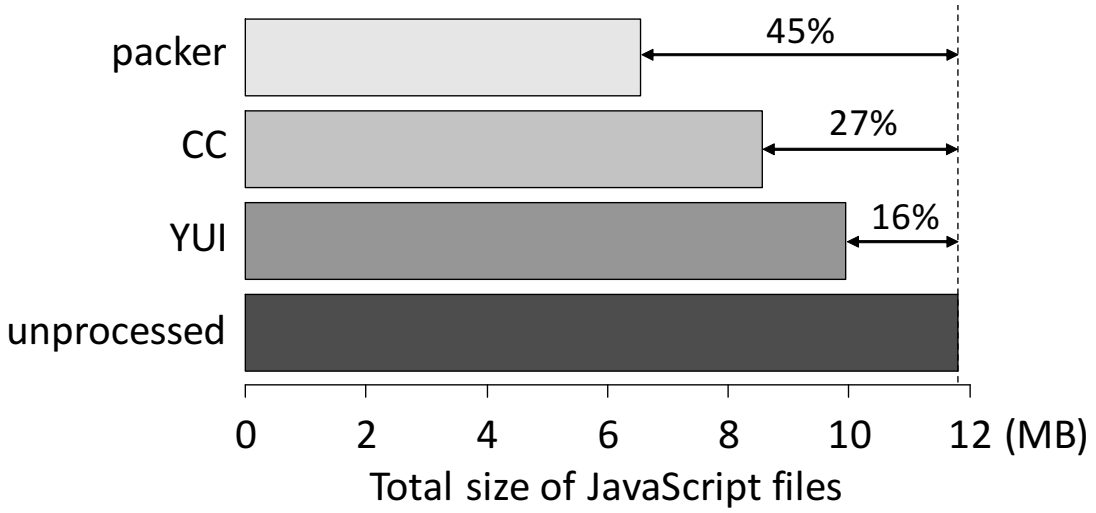


(b) Japanese gov. univ.

Fig. 4.3: Percentages of the usage of $HTTP_{comp}$



(a) Alexa top 500



(b) Japanese gov. univ.

Fig. 4.4: Minification effects of $Script_{opt}$

Table 4.2: Detailed Minification effects of $Script_{opt}$

(a) Alexa top 500

	packer	**CC**	**YUI**
Total JavaScript Size	96.4MB	131.2MB	149.7MB
Reduction JavaScript Size	61.7MB	26.9MB	8.4MB
Optimization Rate	61.0%	83.0%	94.7%
Reduction Rate	39.0%	17.0%	5.3%
Average Processing Time	25.0sec	3.8sec	0.9sec
Successfully Compiled Rate	100%	77.5%	79.3%

(b) Japanese gov. univ.

	packer	CC	YUI
Total JavaScript Size	6.8MB	8.9MB	10.4MB
Reduction JavaScript Size	5.5MB	3.4MB	1.9MB
Optimization Rate	55.3%	72.3%	84.6%
Reduction Rate	44.7%	27.7%	15.4%
Average Processing Time	4.4sec	2.6sec	0.7sec
Successfully Compiled Rate	100%	88.4%	93.0%

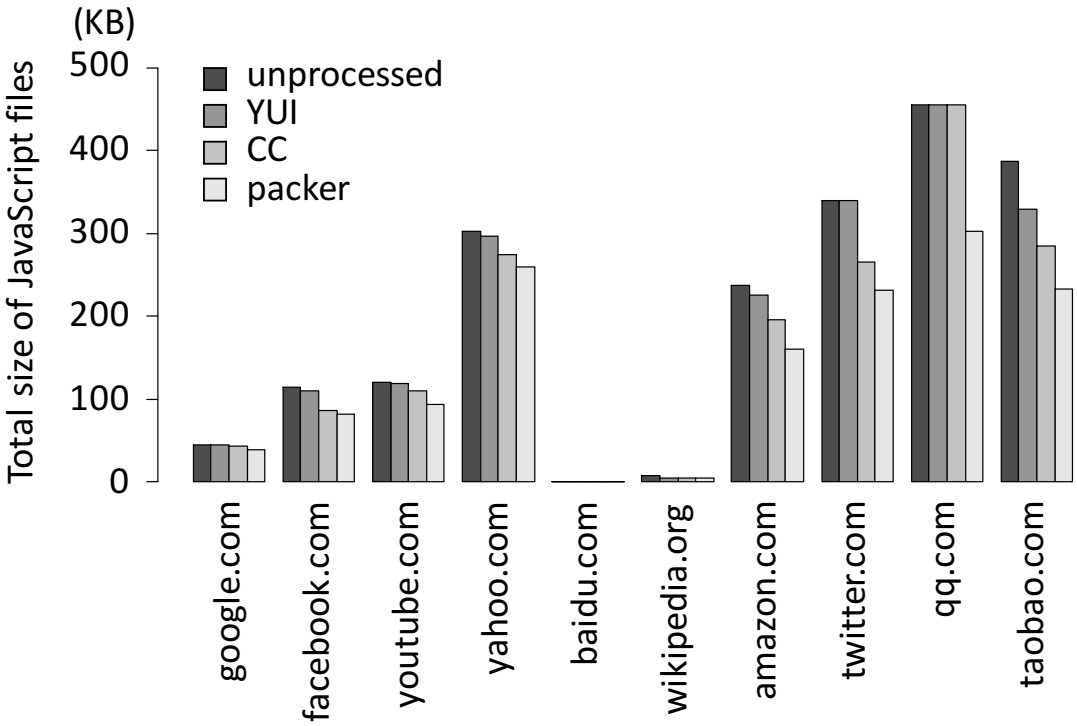
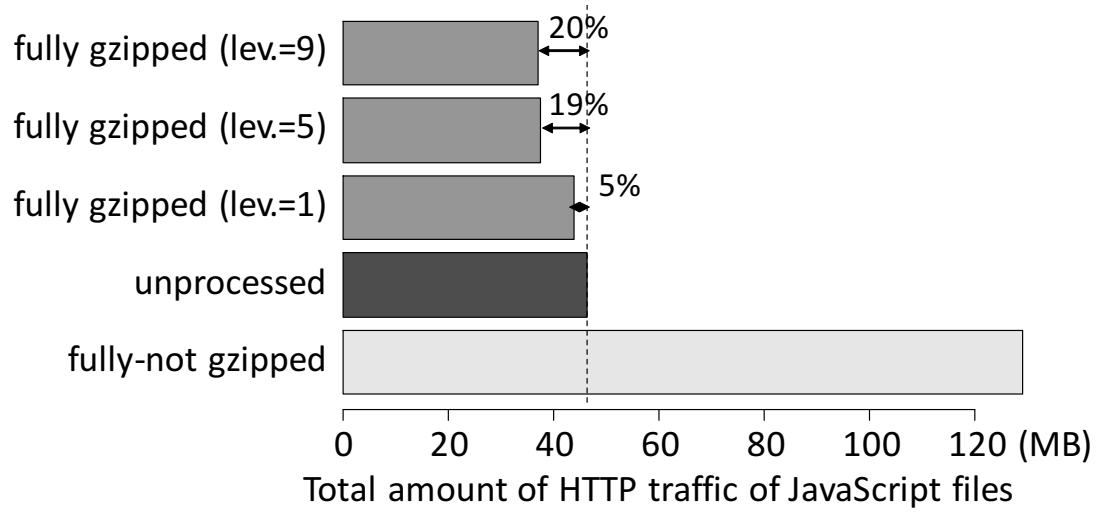
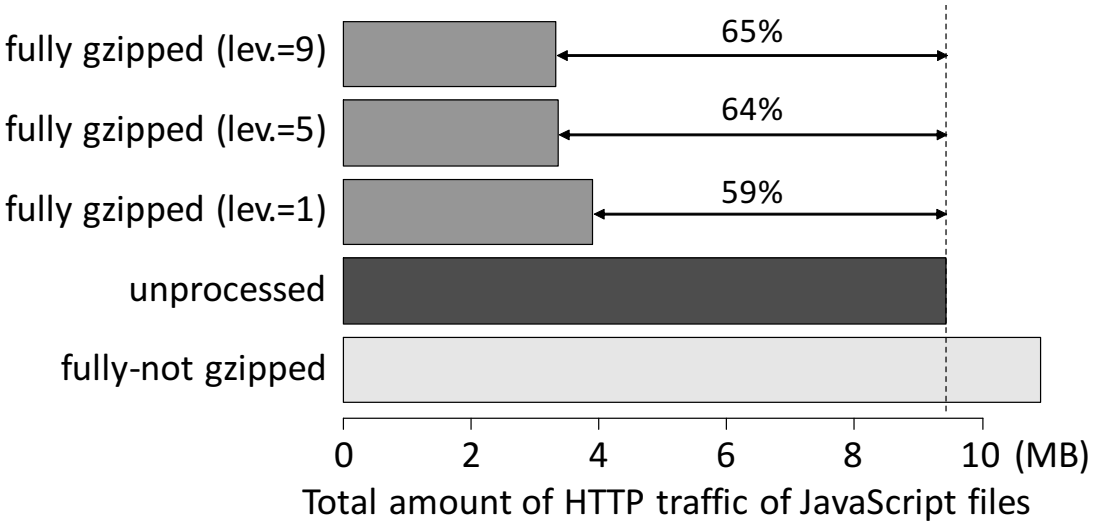


Fig. 4.5: Detailed minification effects for the top 10 websites



(a) Alexa top 500



(b) Japanese gov. univ.

Fig. 4.6: Compression effects of $HTTP_{comp}$

第 5 章

関連研究

Web 上の通信トラヒック削減を目的とした $Script_{opt}$ や $HTTP_{comp}$ の結果は Web コンテンツに含まれている JavaScript の割合に強く依存する．Ihm と Pai は実際の Web トラヒックデータを 5 年間計測・分析し，現在の Web トラヒックについて報告した [5]．彼らの報告により，マルチメディアコンテンツは約 50%（画像：10%，動画：40%），テキストベースのコンテンツは約 45%（HTML：25%，CSS：5%，JavaScript：15%），その他のコンテンツは約 5% トラヒックを消費していることがわかった．よって，本実験は Web 上のトラヒックデータの 15% に効果があると考えられる．

Web コンテンツに最適化・縮小化を行うための様々な技術がある [7][14][15][16][17]．Martin らは JavaScript を抽象構文木に変換する手法で最適化ツールを提案した [15]．また，不要なコードを削除することでデータサイズを削減するという縮小化の考えは，HTML や CSS にも適応できる [16]．いくつかの Web コンテンツを一つの HTML ファイルにまとめると，HTTP リクエストの数を削減できるので，その結果，データ転送量を削減することができる．

Google Chrome のモバイル用の Web ブラウザは通信トラヒックを節約する機能がブラウザに組み込まれている [18]．「データ使用量を節約」の設定を ON に設定すると，全ての HTTP リクエストが Google のプロキシサーバを経由し，テキストベースのコンテンツは最適化で圧縮され，HTTP 圧縮が適用される．さらに，画像ファイルは WebP という画像フォーマットに変換される．Google によると，この設定で Web ページのサイズを 50% 削減することがで

きることが報告されている．

現在提供されている最適化ツールや論文とは対照的に，本研究が貢献している点は，実際に存在する 500Web サイトで実験を行い，削減の余地があることを示していることである．他の最適化技術で行うことは今後の課題である．

第 6 章

まとめ

本稿は 2 つのアプローチ（スクリプト最適化，HTTP 圧縮）に着目し，Web サイトに含まれる JavaScript にサイズ削減の余地があるかを調査した．

結果は以下の通りである．

- Alexa 社が提供しているアクセスランキングトップ 500 の Web サイトでは，87% の Web サイトがスクリプト最適化で JavaScript のサイズを削減する余地があった．
- トップ 500 の Web サイトの 92% がすでに HTTP 圧縮の設定がされている．一方，あまりアクセスされない Web サイトでは，85% の Web サイトで HTTP 圧縮の設定がされていなかった．
- スクリプト最適化で JavaScript のファイルサイズの 39% を削減することができた．
- HTTP 圧縮により，現在 JavaScript ファイルにかかる通信トラヒック量の 50% 以上削減することができている．さらに全ての Web サイトで HTTP 圧縮の設定を行うと，5% から 20% トラヒック量を削減することができる．

謝辞

本研究を進めるにあたり，多くの方々に御指導，御協力をいただきました．ここに感謝の意を表したいと思います．

神戸大学大学院システム情報学研究科 鳩野 逸生 教授，上原 邦昭 教授，並びに羅 志偉 教授 には，本研究の審査過程において有益な御助言と御指導を賜りました．深い感謝の意を申し上げます．

神戸大学大学院システム情報学研究科 中村 匡秀 准教授には，本研究の指導教員を担当していただき，研究に対する指導のみならず，研究室内外における活動においてあらゆる面で熱心な御指導をして頂きました．心より深く感謝申し上げます．

神戸大学大学院システム情報学研究科 柗本 真佑 特命助教 並びに 佐伯 幸郎 特命助教 には，研究に対する取り組み方や姿勢，論文の書き方などを丁寧に御指導して頂きました．心より深く感謝申し上げます．

また，研究室の生活の事務的な面で支えて頂いた神戸大学大学院システム情報学研究科超並列アルゴリズム講座事務補佐員 中村 純子 女史に深く感謝申し上げます．

最後に，神戸大学大学院システム情報学研究科超並列アルゴリズム講座の諸氏には，日頃の研究生活においてさまざまな御助言を頂き，心地よい研究環境を与えて頂きました．心より深く感謝申し上げます．

参考文献

[1] Alex Wright. Ready for a web OS? *Commun. ACM*, 52(12):16–17, 2009.

[2] Aaron Weiss. WebOS: Say goodbye to desktop applications. *netWorker*, 9(4):18–26, 2005.

[3] 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.

[4] S. R. Kodituwakku and U. S. Amarasinghe. Comparison of lossless data compression algorithms for text data. *J. Computer Science and Engineering*, 1(4):416–425, 2004.

[5] Sunghwan Ihm and Vivek S. Pai. Towards understanding modern web traffic. In *Internet Measurement Conference*, pages 295–312, 2011.

[6] Steve Souders. High-performance web sites. *Commun. ACM*, 51(12):36–41, December 2008.

[7] GailRahn Frederick and Rajesh Lal. *Optimizing Mobile Markup*, pages 213–238. Apress, 2009.

[8] Zhigang Liu, Y. Saifullah, M. Greis, and S. Sreemanthula. HTTP compression techniques. In *Wireless Communications and Networking Conference*, volume 4, pages 2495–2500, Mar. 2005.

[9] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Håkon Wium Lie, and Chris Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Conf. Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 155–166, 1997.

[10] Gary A. Kildall. A unified approach to global program optimization. In *Symp. Principles of Programming Languages*, pages 194–206, 1973.

[11] Julien 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] Dean Edwards. A javascript compressor. version 3.0. `http://dean.edwards.name/packer/`. [Online; accessed 01-Dec-2014].

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

[15] Martin Burtscher, Benjamin Livshits, Gaurav Sinha, and Benjamin G. Zorn. JSZap: Compressing javascript code. In *USENIX Conf. Web Application Development*, pages 39–50, 2010.

[16] Andrew B. King. *Website Optimization*. O'Reilly, 2008.

[17] Feng Qian, Junxian Huang, Jeffrey Erman, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. How to reduce smartphone traffic volume by 30%? In *Int'l Conf. Passive and Active Measurement*, pages 42–52, 2013.

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