UNIVERSITY OF CALIFORNIA
SANTA CRUZ

LIVEJS: IMPROVING JAVASCRIPT LOADING TIMES

A dissertation submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Daphne Irene Gorman

June 2016

The Dissertation of
Daphne Irene Gorman is approved:

Professor Jose Renau, Chair

Professor Cormac Flanagan

Professor Jishen Zhao

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

LiveJS: Improving JavaScript Loading Times

by

Daphne Irene Gorman

Despite the prevalence of JavaScript in modern web development, most JavaScript optimizations are designed to improve run-time. However, the load-time of websites is also an important aspect of a user's experience. Therefore, it would be advantageous to have a method for significantly decreasing the amount of time it takes to load webpages.

We propose LiveJS: a tool that utilizes seamless profiling information to modify the order and method JavaScript functions are sent to the client, thus providing a speedup in loading time. The initial JavaScript response LiveJS sends to the user is shorter, as the definitions of most of the JavaScript functions have been removed or delayed. Following that, the rest of the used functions are sent in an asynchronous response from the server. Because the initial file size is smaller, the web browser is able to start compiling and even running the JavaScript code it has before the rest of the data has even finished downloading. The evaluation shows the speedups with highly optimized websites like the Google calendar and the Amazon website, showing an average of more than 20% load time reduction across all benchmarks.

Chapter 1

Introduction

Since its introduction in 1995, JavaScript's ubiquity has increased, and now it is included in almost all modern websites [4,9]. Figure 1.1 shows the top 1000 websites reported by httparchive with the most traffic [22]. Additionally, the average JavaScript code size has nearly tripled in the last 4 years. As such, it would be advantageous to optimize JavaScript usage in order to provide faster services to web users. It follows that there are many tools available to web developers that offer JavaScript optimization and methods to improve JavaScript efficiency. Furthermore, large companies are working towards improving their browsers so that JavaScript will load and run more quickly [14].

To date, most JavaScript optimization tools are aimed at run-time optimizations. Large corporations such as Google and Mozilla have created optimizing JavaScript compilers that allow for load time optimizations like supporting asynchronous JavaScript loads and separated threads for compilation. Additionally, some tools are available to web developers to parallelize and optimize code for run-time [20]. Unfortunately, tools available to web develop-

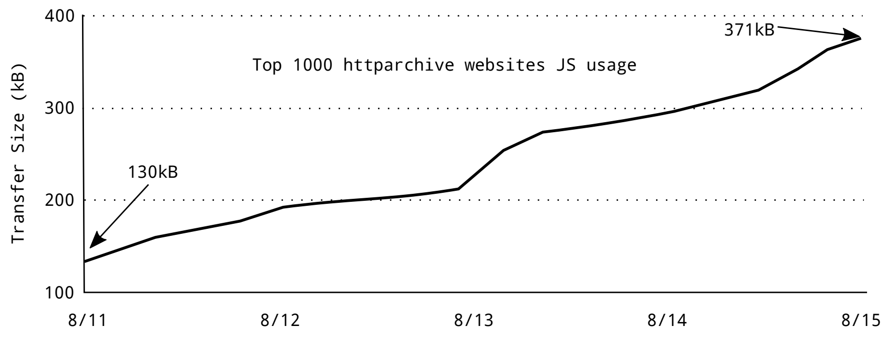

Figure 1.1: JavaScript usage keeps increasing over time. Increasing from 130 kB to 371 kB in the last 4 Years for the 1000 websites with the most traffic [22].

ers that decrease the load-time of webpages are generally restricted to minimizing the amount of code being sent to the client such as the Google Closure compiler [8] and applications such as UglifyJS [2].

In order to load a webpage, it is rarely necessary to utilize all of the JavaScript functions that are provided by the server. Often times, many functions are platform-specific, and thus useless on all other platforms. Furthermore, many functions provide functionality to webpages, but are not needed for rendering the images or populating the content. Some webpages contain dead code, or functions that are never used. Equally as important, many functions are not needed until later in time a special event is triggered such as a mouse click or a keystroke. In Figure 1.2, the percentage of functions that are used by the browser over time is shown for Google Calendar. Even after a minute and a half, less than ten percent of the functions are used by the client.

As Google developers have noted, the speed of compiling and running the actual code is only a component of JavaScript optimization; loading web pages is also a large (and probably more noticeable) part of a web-browsing experience [14]. The aim of this work is to

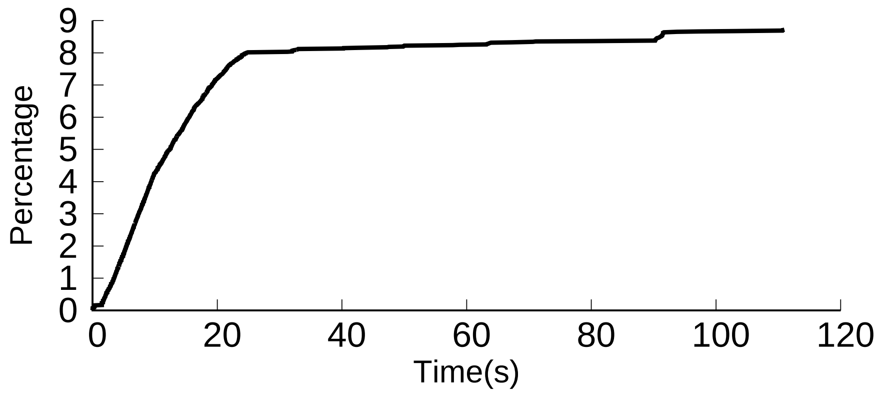

Figure 1.2: Less than 9% of the JavaScript functions are used in the first seconds of execution when using Google Calendar.

provide an orthogonal way to optimize JavaScript load times that is compatible with existing methodologies.

The traditional process for loading a webpage includes three main parts that are performed sequentially [14]. The first part includes downloading the actual web page. This includes downloading all code elements, including HTML as well as JavaScript, as well as pictures, music, and other digital media. Only after each individual file download has completed do traditional browsers start parsing the code. This means that the browser does not start doing anything with the information received from the server until it has the following data in the file whether or not that data is relevant.

The next step in the page loading process is the parsing of the code [14,26]. That is, the JavaScript code is broken down into an Abstract Syntax Tree (AST) that is to be interpreted later on. Once again, during this phase, the user will not be able to utilize the JavaScript code.

The final phase, standard in most web browsers, is the interpretation or code generation of this syntax tree. At this phase, the user is finally able to use the JavaScript that is embedded in their website. An exception is observed in the Google web browsers, that skip this step and compile the parsed code into native machine code [26]. This extra step allows the JavaScript in the website to actually have a faster run time, at the cost of some load time. The Mozilla approach is to use a more traditional Just In Time Compiler (JIT) where code is optimized only after it is interpreted several times [26].

Proposed in this work is a technique called LiveJS that provides an alternative way to optimize the loading time of JavaScript components in web pages and is described more in depth in the following chapters. Briefly, LiveJS is a tool that modifies existing JavaScript code in order to allow the parsing and compilation to begin before the JavaScript file download phase has completed. Each JavaScript file is divided in three categories: inlined, asynchronous, and synchronous. Inlined functions are needed early in the execution time. Synchronous functions are functions frequently used after the execution starts but their transfer can be delayed. Asynchronous functions are rarely used. Inlined functions are send with the original file request, synchronous functions are sent with a separate file immediately afterwards, and asynchronous functions are only sent on request when the associated functions are used.

To the user, the web content will appear to have loaded quicker than if LiveJS had not been utilized. LiveJS has been designed to be compatible with all the current solutions and add further efficiency to the existing optimizations in the browsers.

Using this technique, we were able to get an average of 30% load time reduction in the already well-optimized Google Calendar. Across all benchmarks, we observed an average 21% load time reduction, and in no cases did we observe an increase in loading time. The main contributions of this work include:

- **The proposition of LiveJS** as a tool to decrease the load time of websites. We provide the model for LiveJS and describe the flow for loading a website using the LiveJS tool.
- **A methodology to test the efficacy of such a tool** on multiple webpages and web clients. We describe a test setup that uses real-world websites to compare the load time with and without LiveJS.
- **The evaluation of LiveJS** as a viable tool to attain speedups in real-world websites. We provide an analysis as to why each benchmark was affected differently by the LiveJS tool, and show the overall impact that LiveJS has on website load times.

This work provides the model for the LiveJS tool proposed in Chapter 2. Chapter 3 describes the details for how we set up our experiments. Our results will be provided in Chapter 4. We will describe some current related work in Chapter 5, and finally we will explain our conclusions in Chapter 6.

Chapter 2

LiveJS Model

This chapter provides the structure of LiveJS and describes the inner workings of the tool.

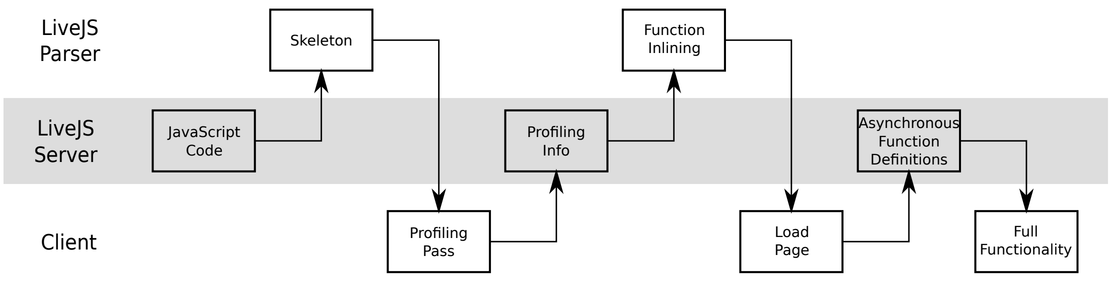

Figure 2.1: Flow of LiveJS in terms of the server, the JavaScript parser, and the client. The server and the parser are LiveJS specific, whereas the client is generic and can be any web browser.

2.1 LiveJS Flow

LiveJS uses profiling information to build a timeline of the JavaScript functions usage. Once the relative JavaScript function order is built, LiveJS can sort the original JavaScript code by function usage in order to discern which functions should be sent to the client first. Figure 2.1 shows the three main components and the steps to achieve a load time optimized JavaScript code: the main server hosting the website, a JavaScript parser and the client loading the website. The parser we utilize is Esprima [13].

When new JavaScript code is provided to the LiveJS server, it is immediately sent to the Esprima parser to create a JavaScript "skeleton". This "skeleton" contains the structure of the original code, but has relatively poor functionality, as the bodies of each function definition have been replaced with a synchronous XMLHttpRequest (XHR) to the server.

In its basic state, the skeleton code is impractical for repeated use. However, every time it receives an XHR, the LiveJS server is able to obtain profiling information. Thus, once this profiling pass is complete, the server is able to utilize its results in order to insert the original function definitions back into the skeleton. During this "inlining phase", the LiveJS server re-inserts the functions that are used first back into the skeleton. Upon completion of sending the file with blank function definitions, the server will asynchronously send the original function definitions in the order that that they are called.

This allows the functions that are necessary early in the execution to be provided in the original script that gets sent when a client loads an optimized webpage. Once that script has finished loading, the rest of the functions the client uses are sent asynchronously to the client in order to provide full functionality. The functions that are rarely used by the client are not sent, so to get them, the client sends one synchronous request per function.

2.2 Load Size Optimization: Skeleton Creation

One of the main bottlenecks of a website's loading time is the size of the file being downloaded. In order to minimize the amount of data, we create a "skeleton" of all the JavaScript that is to be loaded. This skeleton consists of all the functions in the file, but instead of their original definitions, the bodies have all been replaced with our shorter LiveJS code. Additionally, some skeletons will contain a string with the definition for the function `livejs()` at the beginning of the file.

After parsing all JavaScript code with the Esprima parser, we are able to utilize the estraverse tool [24] in Esprima to traverse the AST in order to identify all the function definitions in each script. The body of each function is then replaced with two instructions: an `eval()` statement on the string defining the `livejs()` function and a `livejs()` function call.

The `livejs()` function overwrites its calling function with the original function on the server and returns the result of that function applied on its arguments. The intricacies of this function are described in more detail in 2.2.1.

For the `livejs()` function to be effective, it must have access to all the variables in its calling function's closure. In order to ensure that all of the properties of the `this` object as well as the elements in the current scope are preserved in the `livejs()` function call, it is necessary to define the function inside its calling function. We accomplish this by utilizing an `eval()` call on a string containing the function definition for `livejs()` as shown in Listing 2.1.

When the AST has been completely modified as desired, we utilize the escodegen tool [23] in order to generate relatively optimized JavaScript described by the syntax tree.

```
var livejs_def = function livejs(name, scope){
    //livejs function definition
}.toString();

. . .

function example () {
    eval(livejs_def);
    livejs("example", this);
}

. . .

var xhr = new XMLHttpRequest();
xhr.open("GET", async_function_defs, true);
xhr.onload = function (e) {
    eval(xhr.responseText);
}
xhr.send();
```

Listing 2.1: Pseudocode for the layout of the code skeleton including a skeleton function definition example and an asynchronous request at the end of the file.

2.2.1 LiveJS Function

This function ultimately replaces the calling function with its original version: what we had originally replaced with our blank is now once again the function definition.

In the case that this function was being called to create an object, all the functions in the prototype would also be replaced with their original functions before creating and returning the new object.

In order to accomplish this, the `livejs()` function performs a number of steps. The first thing it determines is whether or not the calling function is creating a new instance of an object. If so, we must replace the object that has already been created with a new object, described by the original code (as opposed to the skeleton dummy code). It then determines all the attributes of the calling function: Object prototype definitions and/or other Object keys. Following that, the function sends a synchronous XHR to the LiveJS server asking for the original function definition.

Once the original function definition is received from the server, the `livejs()` function will overwrite its calling function as such. Then it will repopulate that function's keys and prototype attributes. If the keys or the prototype attributes are functions still described by their dummy skeleton definition, `livejs()` will recursively call itself to obtain the original function definitions for those functions. Finally, it will return the newly described function applied on the `this` object provided by the calling function with the arguments provided to the calling function. The pseudocode for the function is provided in Listing 2.2.

```
function livejs(fcn, scope) {
    this.attributes = attributes_from_server;
    this.prototype = prototype_from_server;
    if(this == new Object){
        this = this.prototype();
    }
    return this;
}
```

Listing 2.2: Pseudocode for the `livejs()` function. In JavaScript, the prototype of a function is itself, and the prototype of an Object is its constructor function.

2.2.2 JavaScript Modification

After using Esprima to generate an AST for the original code, LiveJS performs multiple passes to modify the JavaScript. The first three passes format the AST so that all of the functions are positioned in the code such that when the dummy skeleton code replaces the original function body, if the `livejs()` function is called then its calling function can be overwritten in its scope as well as, if necessary, globally. The fourth and final pass performs the replacements of the non-inlined functions with the dummy function on this modified AST.

The Esprima project includes a code generation tool called Escodegen [23], which generates JavaScript based on the AST created by Esprima. LiveJS utilizes Escodegen on the AST of the skeleton code to produce minimized code. The generated code uses all of the optimizations available in the Escodegen tool.

2.3 Load Time Optimization

Using some profiling information, LiveJS is able to determine the appropriate method for providing the client with different JavaScript functions. There are three modes in which JavaScript functions are provided to the client: inline with the original JavaScript file, sent asynchronously after that file has been run, and synchronously only on a `livejs()` function call. As the synchronous function downloads are inherent in the code "skeleton" that is originally created, this section will focus on the inline and asynchronous function loads.

2.3.1 Function Inlining

Some JavaScript functions are used as soon as they are defined, and therefore must be sent as soon as they are requested in order to prevent delays. Since these functions must be defined early, LiveJS will replace the "skeleton" body of those functions with the original function definition, leaving the function inline in the script. By keeping the inlined functions to a minimum, LiveJS is able to reduce the size of the original file sent to the client. Therefore, the browser will be able to begin running and/or compiling the necessary code before the rest of the functions are sent from the server.

2.3.2 Asynchronous Load

Appended to the end of the original file is an instruction to send an asynchronous request to the server for the remaining functions that are used.

Many JavaScript functions in a webpage are utilized only after a page has been fully rendered, and only provide functionality for the page. As such, they are necessary for the page to run, but are not essential for the user to feel as though the page has been fully loaded. Examples of website processes used by such functions include populating rarely-used drop-down menus and hover text.

As such, these functions can be provided after the website is already running. Therefore, LiveJS omits sending these functions with the original file, and only sends them asynchronously, once the original file has been completely processed, during the asynchronous request.

2.4 Profiling

In order to find the relative order by which functions are called and which functions are not frequently called, LiveJS employs some primitive profiling techniques.

The first time a website is used, LiveJS will send only the "skeleton" to the client. As the skeleton sends a synchronous request to the server for every function being called, LiveJS will be able to determine which functions are used. Additionally, by taking into account the order in which it receives XHRs, the LiveJS server is able to procure a general ordering for when the functions are first needed. The reason is that this is a simpler profiling than typical, us that LiveJS only takes into account the relative order of functions being called, not the execution time of functions.

Using this information, LiveJS is able to determine which functions to provide inline with the file as well as which functions will be sent asynchronously.

2.4.1 Profile Gathering

LiveJS keeps track of a general order in which JavaScript functions are first used. As opposed to most run-time JavaScript optimizations, LiveJS does not need to take into account a function's frequency of use and therefore avoids any associated overhead. Thus, keeping only a basic ordering in which the functions are used is sufficient for LiveJS to perform as desired.

Obtaining this information is done in two ways: once the first time a webpage is loaded in an initial profiling pass, and also during every subsequent page load when the `livejs()` function is called.

When a LiveJS server is first set up, it will lack any profiling information, and thus the first client to request a webpage will be provided with only the JavaScript skeleton. As the skeleton contains none of the original function definitions, in order to run the webpage, each time a function is used, its `livejs()` function will be called, triggering a synchronous request to the server. The LiveJS server will keep track of the order in which these requests are received, and therefore be able to monitor the order each function is actually called.

After that original profiling pass, LiveJS continues to obtain profiling information. If a function is requested synchronously, LiveJS knows that somehow that specific function has not been adequately provided to the client in time. Thus, that function is given a higher priority to be sent earlier. If that function has never been used before, the next time that webpage is loaded, it will be sent asynchronously. If that function was originally sent asynchronously, it would be inlined the next time the webpage is loaded.

In order to keep the LiveJS statistics and function call order up to date, periodically the

LiveJS server will insert some asynchronous XHRs (a simple POST request with the function ID) into a small percentage of the inlined and asynchronous function definitions in order to keep track of the relative ordering in which clients utilize different functions. This allows the LiveJS server to maintain a current function order by getting information from multiple clients.

2.4.2 Profile Usage

Depending on what type of network a client connects from, a LiveJS server will be able to provide a function breakdown that is optimized for that connectivity. As different websites utilize JavaScript functions in different ways, it might be advantageous for one website to inline more functions for a slow connection, but send more in the asynchronous reply for fast connections, whereas other websites may experience the opposite. A more in-depth description of these cases are provided in Chapter 4.3.

Therefore LiveJS servers will keep track of when synchronous requests for function definitions are received from different network types and modify the ratio of inlined to asynchronous functions accordingly per bandwidth.

Chapter 3

Measurement Setup

This chapter describes our strategy for testing the speedup LiveJS provides. In order to test LiveJS on widely used websites, we implemented a test setup using a proxy server, a node.js server, and a web browser. All tests were performed on an i7 processor running Arch Linux.

3.1 Proxy Server

In order to determine the efficacy of LiveJS, we implemented a proxy server using the mitmdump apparatus in the mitmproxy tool [7], which allows us to set up a man-in-the-middle proxy server that runs locally on the same machine as the client, intercepting all data transfer between a browser and any servers. Using mitmdump, we were able to test the impact LiveJS has on real websites such as Google Calendar and Amazon.

Utilizing mitmdump, we were able to intercept all the responses from the server and apply our LiveJS tool. We first identified all the server responses that contained JavaScript, such as a JavaScript file or an HTML document containing script tags. From there, we were able to isolate all the JavaScript code and run it through our Esprima parser described in Chapter 3.3.

As the proxy server intercepts all data being transferred to and from the server, we were able to store all the scripts that were being sent to the browser. Once we obtained all the code, our proxy server was able to function as though it was the original server: it can handle all the requests locally and therefore completely cut out all communication with the original server.

When a request for new JavaScript code is received from the client, our proxy server determines whether or not that code has already been stored locally. If not, it forwards the request to the server and waits for the response. Upon receiving the response, it sends all of the JavaScript code to an Esprima parser that works as described in Chapter 2 and then receives a skeleton and a list of all the function definitions. It also saves the original JavaScript file. The proxy server sends the skeleton to the client, and monitors every request for a function definition. In doing so, it is able to maintain a priority ordering for when the functions should be sent (based on the order in which they are used). As the proxy server also has all of the function definitions, it is able to send the JavaScript function definitions directly, without needing to further utilize the Esprima parser until the next time the webpage is loaded.

If the proxy server receives a request for JavaScript code it has encountered previously, then it sends the saved JavaScript code to the Esprima parser and requests a skeleton that has some inlined functions and includes an asynchronous request.

As our proxy server is standing in for an independent LiveJS server, it keeps track of all the profiling information. Upon receiving a synchronous request from the `livejs()`

function call, the proxy server will identify the function being requested and update its priority. If the function has never been used before, it will assign the next available priority to that function. If the function has been used, it will assign a higher priority to that function.

Depending on a function's priority, the proxy server is able to determine the order in which functions are provided to the client and by what means. When the proxy server receives a synchronous request for a function definition, it determines whether or not that function has been previously sent asynchronously based on its current priority. If it has not, that function will be set asynchronously the next time the web page is loaded. If the function has been sent asynchronously, the proxy server will increase the number of functions that should be sent inline, and assign that function a higher priority for getting sent first. Therefore, the next time the page is loaded, that function will have a higher probability of being sent inlined with the initial JavaScript response.

3.2 Client Setup

We used the Chromium browser version 49 and Firefox version 45 in order to test LiveJS. Chromium includes developer tools that allow users to test their websites on different platforms and have an option for limiting bandwidth. On the other hand, to conduct the tests in Firefox, we utilized The Wonder Shaper 1.2 [15]. The differences in the results we obtained from each browser was negligible (within a 2% error margin), and thus this document will report the results obtained by the Chromium client.

Using the throttling tools, we were able to test LiveJS on websites as though the user

Table 3.1: This table shows the speed and delay of the networks used during testing.

	2G	3G	4G
Speed	250 Kbps	750 Kbps	4 Mbps
Round-Trip Delay	300 ms	100 ms	20 ms

is accessing them through 2G, 3G, and 4G networks as well as turning off bandwidth throttling entirely. As all of the data transfer was performed between the client and our proxy server, we were able to perform our tests without worrying about the connectivity with the actual server, and thus the connection speed selected in Chromium was the determining factor for bandwidth. The exact speeds for each network are described in Table 3.1.

Other than throttling the bandwidth for testing different network speeds, we made no changes to the client test setup. Thus, we are able to test LiveJS as a tool for web developers that is not impacted by the client.

3.3 Parsing JavaScript

In order to test LiveJS on widely-used websites, we implemented our technique using the Esprima JavaScript parser running on a node.js server.

Upon intercepting JavaScript code, the proxy server in our test setup would send a request for a JavaScript skeleton for that script to the node.js server running the Esprima parser. When it receives a request for a skeleton, the node.js server simply parses the script and replaces the body of each JavaScript function as described in Chapter 2.2.

When the proxy server receives a request for a script that it has already stored, it will send a request to the node.js server for a JavaScript file that has inlined functions. This request will provide the original function definitions and some profiling information including function priority (or a general order) as well as the percentages of functions that should be sent inlined with the file as well as asynchronously. As before, the Esprima parser will isolate all the functions definitions. However, this time, it will only replace the functions that should not be inlined with their dummy skeleton definitions. Additionally, the parser will also append a server request at the end of the file to receive all the functions that are sent asynchronously.

However, the node.js parser will not provide a function definition for a function that has never been used. If the proxy server requests a higher percentage of functions to be inlined than there are used functions, the file will only inline the functions used and no functions will be sent asynchronously. Additionally, if the sum of the percentages requested for inline and asynchronous functions exceeds the number of functions used, the parser will inline the requested percentage of functions, but less functions will be sent asynchronously.

3.4 Benchmarks

In order to test LiveJS, we were able to utilize our testing setup on six real-world websites: Google Calendar, Amazon.com, OS.js, Facebook.com, CNN.com, and Twitter.com.

3.4.1 Google Calendar

One of the benchmarks we used to test the efficacy of LiveJS was Google Calendar.

The original version of this website (before being modified by the LiveJS setup) is already highly optimized. When tested, it is apparent that the JavaScript code is split up into multiple files, and the first file sent contains the functions that tend to be used first. Therefore, we utilized this benchmark in order to determine the impact LiveJS would have on code that already has ordering taken into account.

A majority of the Google Calendar website's code is JavaScript. Of the 3.7 MB of data that gets sent from the server to the client, 3.5 MB (95%) is JavaScript code. Most of the code for rendering the page is done in JavaScript, with very little CSS code. Additionally, all of the events in the calendar are obtained using JavaScript code. Therefore, utilizing a JavaScript optimization tool should make the website faster.

3.4.2 Amazon.com

In order to test LiveJS on websites that are widely used, we used Amazon.com as a benchmark. Amazon.com uses JavaScript in multiple ways such as populating the page based on each user's preference as well as powering its search engine. However, a majority of the data being transferred to the client from the Amazon.com server is actually not JavaScript code, but rather images. Out of the full 10.1 MB, only 7% of the data is scripts.

We utilized this website to determine the effect of LiveJS on websites that have a small percentage of JavaScript code.

3.4.3 OS.js

The third benchmark we tested was the OS.js project: a JavaScript Desktop Platform that runs entirely on the web. OS.js includes a window manager and several applications.

Most of OS.js's data is JavaScript (66% of the total 708 KB). The next significant portion of data is once again images, as with Amazon.com. However, in OS.js, there is a single image - the desktop wallpaper - that takes up 14% of the data on its own, with the other images being relatively negligible in size, comprising a total of about 6% of the total data. Thus, all the data for the wallpaper image will be sent all together, instead of in many different (potentially parallelized) server responses as with Amazon.com.

3.4.4 Facebook.com

In order to observe how LiveJS performs on a large social network site, we used Facebook.com as a benchmark. Facebook has a large mix of different types of code and images, and JavaScript does not comprise the majority of Facebook's data on a load, only about 50% of the full 2 MB. Instead, Facebook contains many images as well as CSS code.

3.4.5 CNN.com

CNN.com was used as a benchmark because not only is it widely used around the globe, it also is rendered using JavaScript in order to maintain consistent appearance and functionality across mobile and desktop clients. Additionally, the website has been subject to critiques claiming the current version, although more aesthetically pleasing, is now slower to load [3, 12]. In total, the JavaScript code for CNN.com takes up 55% of the total website data (5 MB).

As a popular website that relies heavily on JavaScript during load-time, CNN.com was an obvious choice for testing LiveJS.

3.4.6 Twitter.com

Twitter.com was used as a benchmark in order to test LiveJS as it is widely used but has relatively minimal functionality. Mostly content and images, Twitter.com's 4 MB is only about 15% JavaScript, most of which is JQuery definitions and analytics for site traffic.

Chapter 4

Evaluation

This chapter will describe the results we observed and provide an evaluation of our findings. As the results from Chromium and Firefox were comparable, only the Chromium results are reported in this chapter.

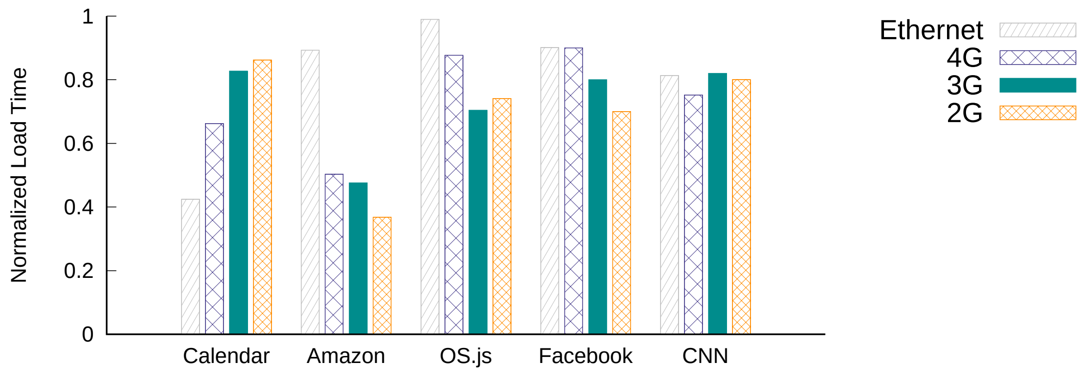

Figure 4.1: Normalized Load time for LiveJS for each benchmark with respect to load time without using LiveJS. The bandwidths are as described in Table 3.1

4.1 Profiling

The algorithm LiveJS used to determine which functions will be sent with the original JavaScript response and which functions will be sent asynchronously depends heavily on the initial profiling pass. Potentially, LiveJS will have poor performance for the next few times the page is loaded, but the algorithm stabilizes quickly and will provide the client with function definitions in a more optimized order. However, once that profiling pass is done, the user will rarely experience an effect from the profiling.

If LiveJS is collecting some profiling data, that means that a function has been called before the server has provided its definition, and thus a synchronous request has been triggered. However, with the exception of during the profiling pass and potentially the first few page loads, these synchronous requests are relatively rare as the LiveJS server will have found a setup in which the appropriate functions are sent inline with the skeleton and asynchronously to the client as to avoid triggering the `livejs()` function. We found that a maximum of around ten page loads was sufficient for the LiveJS algorithm to maintain a workable ordering that mitigates synchronous function calls. This amount of page loads is negligible to websites with a lot of traffic, and can even be performed by the developer before launch.

All the results provided in this chapter were obtained after the profiling passes are complete, and thus are representative of the typical LiveJS behavior.

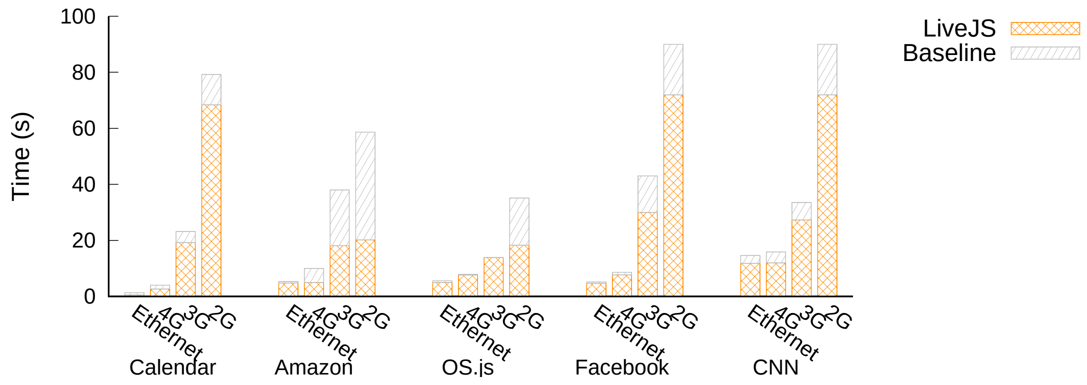

Figure 4.2: Total time taken for each benchmark with and without utilizing LiveJS

4.2 Load Time Decreases

Ultimately, LiveJS is a tool designed to improve the load time for webpages. The results of our testing load times, normalized with respect to the load time without using LiveJS, are shown in Figure 4.1. This section will provide descriptions of the load time decreases as raw data, and analyses of these results will be presented in Section 4.3.

4.2.1 Google Calendar

We were able to achieve a 58% load time reduction for Google Calendar when on a network with unlimited bandwidth. On slower networks, we achieved less speedup but still maintained a faster loading time than the baseline, with the smallest reduction of 14% for a 2G connection speed.

4.2.2 Amazon.com

Amazon.com experienced a more distinct speedup when the network was slower. On the 2G network, the load time was only 37% of the baseline when using LiveJS, but as the connection speed increased, the speedup from LiveJS became less significant. However, for the 3G and 4G networks, the LiveJS load time was still about 2 times faster than the baseline. Even with unlimited bandwidth, the LiveJS provided a 11% load time reduction.

4.2.3 OS.js

For the faster networks, OS.js failed to achieve a significant speedup from LiveJS, with only 1% load time reduction for ethernet speeds and 12% for the 4G connection, but on the 3G network it loaded in only 70% of the time for the baseline, and maintained a significant 25% reduction at 2G speeds.

4.2.4 Facebook.com

When loading Facebook.com, we observed a higher speedup as the network speed declined, loading 30% faster at 2G connection speeds. Without throttling the connection speed, Facebook.com loaded only 10% faster with LiveJS than its baseline time.

4.2.5 CNN.com

CNN.com observed the most significant reduction in loading time from LiveJS with the 3G network connection: 25%. Still, LiveJS provided at at least a 20% reduction consistently at all four connection speeds.

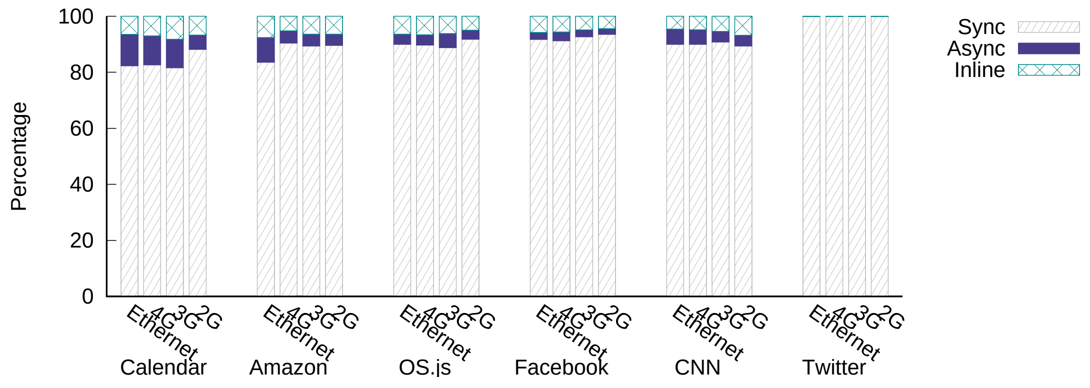

Figure 4.3: Breakdown of functions for each benchmark. The majority of the code remains unused for all three benchmarks, and therefore is sent using synchronous requests. In general, most of the functions that are used are sent inline with the original JavaScript response, and a small percent of functions are sent asynchronously to the client later.

4.2.6 Twitter.com

The amount of speedup provided by LiveJS for Twitter.com was negligible. The load times between the LiveJS optimization and the baseline were the same for all connection speeds. Further explanation will be provided in Section 4.3.

4.3 Insights

This section seeks to provide explanations for the results of our tests. In Figure 4.3, we can see the breakdown of how the functions are sent during a page load.

4.3.1 Google Calendar

Although Google Calendar had the highest percentage of JavaScript functions, LiveJS did not necessarily provide a higher speedup than for the other benchmarks. Because the LiveJS

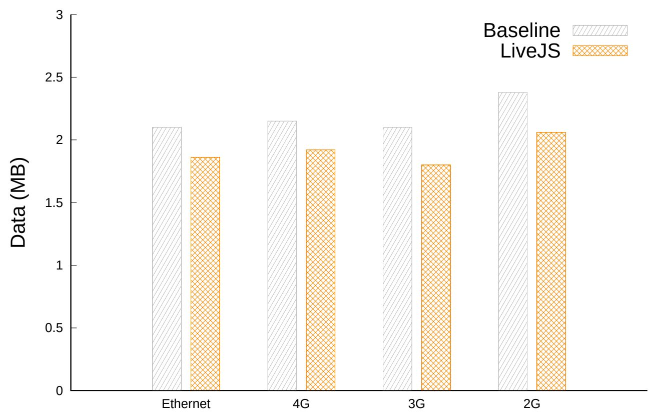

Figure 4.4: Total data transferred from the server to the client when loading Google Calendar. The LiveJS server ultimately sends less data to the client than the baseline.

skeleton still provides the full structure of the original code, it still sends an average of 85% of the amount of data to the client, as shown in Figure 4.4. However, this data is split up into many different files, all of which must be requested by the client. The way Google Calendar is set up the JavaScript files are almost always requested using another JavaScript function. Therefore, if each file can start running an average of 15% faster, then the time saved compounds on it self, giving us the 58% time reduction we observe.

4.3.2 Amazon.com

Of the six benchmarks, Amazon.com was affected most by utilizing LiveJS. Because it relies so heavily on JavaScript in order to populate the website, when LiveJS is used, some of the hidden content is never downloaded from the server. For example, when using a basic setup to load Amazon.com, the client will download a lot of data (mostly images) that may never be displayed on the screen, as they are hidden in menus, or at the bottom of the page. Also, many images downloaded by a web browser are used to populate graphic elements that scroll through

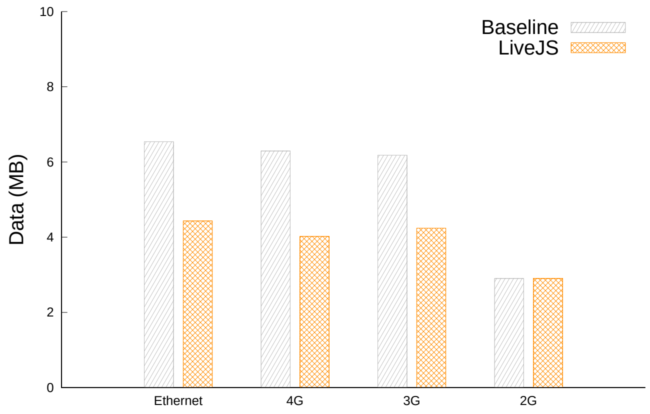

Figure 4.5: Data necessary to load Amazon.com. In general, the LiveJS server would send less data than the baseline.

multiple images, but only display some of them. When using LiveJS, the hidden images are not requested until after the initial page has been loaded and the browser is already running the functions that have been sent asynchronously.

Taking this into account, it makes sense that less data is transferred when using LiveJS to load Amazon.com. Therefore we can justify the observed average reduction of 44% while noting that minimizing the amount of data downloaded also lessens the time it takes before that data can be utilized.

4.3.3 OS.js

The main bottleneck for the OS.js benchmark is downloading a single huge image that acts as the background for the virtual desktop. Furthermore, OS.js effectively only utilizes five JavaScript files. Therefore, although much of the OS.js functionality is implemented in JavaScript, that functionality is relatively unnecessary until after the page as loaded. Figure 4.3 shows the percentages of functions that are sent inline with the skeleton, on an asynchronous

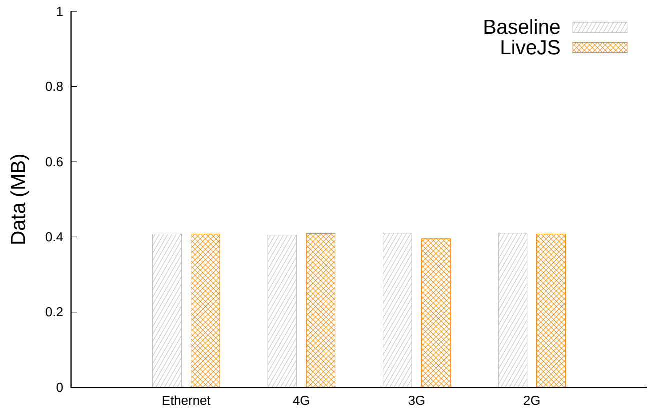

Figure 4.6: Data for loading OS.js. There was a negligible difference in the amount of data sent for the baseline and using LiveJS.

request, or when a function is called and a synchronous request is triggered. As shown in the figure, much of the JavaScript functions are unused when loading OS.js, and are therefore left to be downloaded on a synchronous request. However, as shown in Figure 4.6, generally the same amount of data is downloaded from the baseline and our LiveJS server. This is because, unlike for Amazon.com, OS.js's non-JavaScript data is all immediately visible and used right away. Therefore, LiveJS does not provide as much speedup with OS.js as it does with Amazon.com or Google Calendar.

4.3.4 Facebook.com

The speedup provided by LiveJS for Facebook.com is similar to, but not as drastic, as the speedup observed for the Amazon.com benchmark. Comparing the amount of data transferred during loads of Facebook.com in Figure 4.7 and for Amazon.com in Figure 4.5 as well as observing the reduction in load times for both websites in Figure 4.1, shows similar behavior between the two. Facebook.com also relies on JavaScript in order to populate the website, and

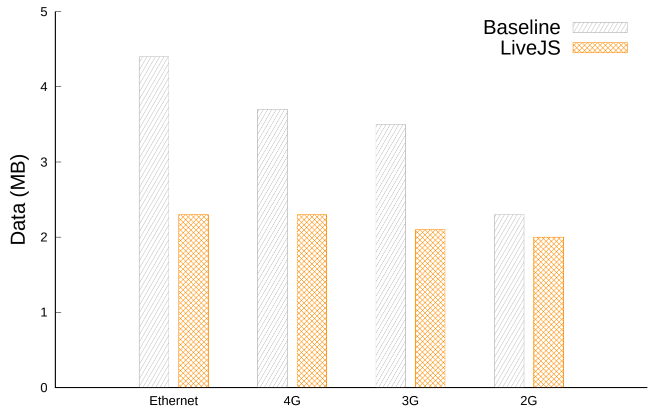

Figure 4.7: Total data transfer while loading Facebook.com. As the network speed increased, there was a greater difference between the amount loaded using LiveJS and without.

thus any rarely-used functionality will never be downloaded. However, unlike Amazon.com which encourages navigation between many separate pages, the Facebook webpage is designed such that a user spends a long time on a single page, with functionality to interact with each element - such as “liking” or commenting - as well as embedded pop-up windows. Since much of the same functionality, and therefore the same code, is used on the readily available elements as well as the hidden or unimportant elements, the code must be available early. Thus, it is intuitive that the LiveJS speedup for Facebook.com is less significant than that for Amazon.com, but manifests in a similar manner.

4.3.5 CNN.com

Using LiveJS with CNN.com provided a load time that was least 20% faster for all four bandwidths, and the speedup remained a relatively consistent percentage. Upon observation, we can determine that the speedup comes from delaying the download of hidden images and functionality from the webpage. As mentioned before, much of the data downloaded by

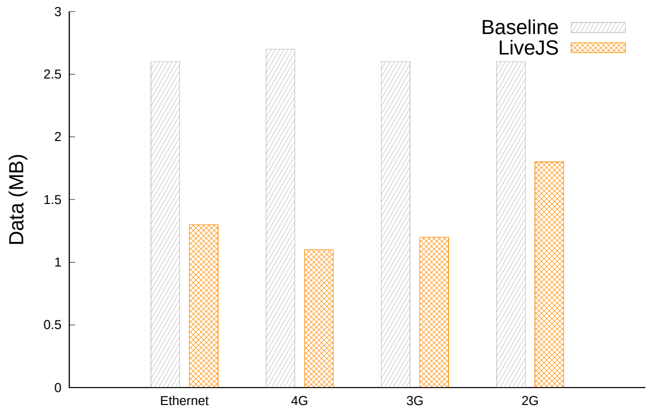

Figure 4.8: Total data downloaded from the server when loading CNN.com with and without LiveJS. When using LiveJS, less data is transferred than the baseline.

CNN.com is JavaScript used for the initial page render, but the website also features many large images that limit how much speedup LiveJS can provide without sacrificing aesthetics. Furthermore, we can determine that the speedup remains consistent across connection speeds due to the fact that the functions get split up for transmission to the client in the same way for each bandwidth, as shown in Figure 4.3. This is because the way CNN.com is designed, the JavaScript functions used for rendering do not overlap with the functions used for functionality, so there is a clear distinction between which functions should be sent inline with the original response and which functions should be sent asynchronously later. The amount of data transferred when loading CNN.com is shown in Figure 4.8.

4.3.6 Twitter.com

The lack of speedup provided by LiveJS for Twitter.com was a direct result of how the webpage is rendered. The JavaScript code for the original website is not used to render the webpage, and thus the download of the JavaScript code can occur in parallel with the other content. Therefore, although LiveJS does not inhibit the load time of Twitter.com, it does not provide any improvement in load times.

Chapter 5

Related Work

This chapter will describe the current work done for JavaScript optimizations. A majority of JavaScript optimizations have been done to improve the run-time, the most notable of which is compiling the scripts and running them natively instead of using an interpreter. However, recently some work has been done to improve load-time as well.

Most load time optimizations have been done in an attempt to minimize the total amount of data being sent for the same amount of code such as shortening the names of the functions and variables [2], or compressing objects before transmission [1]. However, some work has been done in parallelization [16, 20] and inlining [17] of JavaScript downloads.

Recently, Google has succeeded in improving the load-time of websites using JavaScript. These optimizations are both done on the client-side and were released in recent Chrome updates [14]. In March, 2015, they added two new techniques: Script Streaming and Code Caching. Additionally, in early 2016, some work was done on load-time dependencies by MIT and Harvard [18].

5.1 JavaScript Dependency Tracking: Polaris

Polaris is a client-side scheduler designed to modify normal webpages in order to determine which objects to load using a fine-grained dependency graph [18]. Using a dependency tracker subjects Polaris to certain constraints; it cannot support certain JavaScript function calls such as `eval()` and `with()`. As LiveJS does not utilize a dependency tracker, these JavaScript features remain available to developers.

Additionally, Polaris focuses on the reordering of JavaScript between multiple files, whereas LiveJS only reorders the functions contained within a single file. As such, these two tools can work in tandem, providing further speedups.

5.2 Script Inlining: Silo

Silo looks to aggressively inline JavaScript code in order to minimize HTTP fetches and improve the loading time of webpages [17]. This tool aims to download the script content as early as possible by reordering all the website code including HTML and CSS.

Although Silo aims to minimize fetches for load time improvements and LiveJS performs additional fetches to the same end, both tools are actually compatible. If the LiveJS tool is used on a website, the total amount of JavaScript per .js file sent to the client is decreased. Thus, if the Silo framework is used after utilizing LiveJS, the amount of script data being inlined by Silo has also decreased, and so the speedup provided by Silo would also increase. Additionally, the XHRs for the asynchronous and synchronous function loads from LiveJS would still be functional, and therefore the improvements from LiveJS would remain without sacrificing functionality.

5.3 Parallelization of Downloads

One tool available to web developers currently is HeadJS, which looks to optimize websites [20]. HeadJS focuses on improving the load time of websites by loading the JavaScript asynchronously in parallel while still executing all the functions in order. Additionally, it attempts to universalize websites by determining the browser type of the client, and applying dedicated JavaScript logic for that specific platform.

HeadJS is an optimization that allows developers to write lighter code, as this plugin deals with the browser-specific intricacies. Therefore, there is less total JavaScript to download when loading the webpage, and thus the webpage will have a shorter load time. Additionally, with the parallel download capabilities, HeadJS provides a reasonable speedup for JavaScript website load times.

The parallelization of JavaScript downloads is reminiscent of the asynchronous XHR in LiveJS, but ultimately the web developer is accountable for implementing this functionality in HeadJS, whereas LiveJS will perform its optimizations automatically. Additionally, the LiveJS functionality should be unaffected by the parallelizations available in HeadJS, and thus the two tools can be used together for greater improvements.

5.4 JavaScript Code Size Reduction

UglifyJS [2], JSCompress [25], and JavaScript Minifier [5] are just a few of the numerous tools available to web developers that reduce the size of their JavaScript code. Essentially, all of these tools run the JavaScript code through a parser and rename the variables and functions to short names and/or remove extra white space or unnecessary symbols without changing the shape of the AST and thus any of the functionality or even the flow of the code. That is, these tools only remove unnecessary characters in the source code.

Google Closure Compiler: Developed by Google engineers, the Closure Compiler is an application designed for web developers to optimize their code [8]. The idea is that the developers will run their code through the Closure Compiler, and the code will be made more streamlined. This allows for less code volume (as all the variables names have been shortened, etc), and therefore less code to download at load time. Cutting down the download time will effectively shorten the load time of a webpage. The Closure Compiler boasts of helping the developer further by providing warnings for illegal JavaScript code or dangerous operations. However, it is ultimately similar to previous projects such as UglifyJS and others.

When rebuilding the JavaScript code before sending it to the client, LiveJS uses the size reduction techniques available in Escodegen that are comparable to an UglifyJS type optimization in addition to its own unique reordering techniques.

5.5 Google Script Streaming

With the addition of Script Streaming in Chrome version 41, the browser now uses two threads for page loads: a main thread and a parser thread [14]. As previously, the main thread begins downloading the JavaScript code from the server, but now as the main thread is downloading, the parser thread starts parsing the downloaded code. As the download occurs, the parser thread only works on asynchronous and deferred scripts, but when the download completes, the parser finishes all of the code. Once all the code has been parsed, the parser thread is done and the main thread resumes, beginning with the compile phase and continuing with execution.

As we used a newer version of the Google web browser, all of the results in this work utilize this technique for both LiveJS and the baseline measurements. Thus, all the improvements shown are in addition to the benefits of Script Streaming.

5.6 Google Code Caching

In version 42, Chrome started performing code caching as well as script streaming [14]. Code caching focuses on speeding up page loads for repeated visits to the same page. In previous versions, Chrome discarded the compiled JavaScript code for a webpage once a user had navigated away. However, with the introduction of code caching, Chrome now stores a local copy of the compiled code in order to skip the download, parse, and compile phases of a page load. This saves about forty percent of compile time across all page loads.

LiveJS provides load time improvements the first time a webpage is loaded, which is impossible to do with a caching technique. However, all of the caching benefits would still apply to a LiveJS website, as all of the necessary data is eventually sent to the web browser and therefore can be cached.

5.7 Runtime Optimizations

Most of the JavaScript optimization work has been done for run-time. Large corporations such as Mozilla and Google are releasing browsers that incorporate many run-time optimizations, but there are also tools that assist web developers in optimizing their code as well. [6, 10, 11, 19, 21, 26].

As LiveJS is aimed at load time optimizations while still keeping the functionality the same, LiveJS's benefits do not affect the runtime optimizations.

Chapter 6

Conclusions and Future Work

As the evaluation shows, LiveJS is able to speedup heavily optimized websites like Amazon and Google Calendar. Using the LiveJS test setup, we were able to observe a speedup in every benchmark we tested, including heavily optimized websites, as well as over many different bandwidth speeds using multiple different web browsers. LiveJS provides an average speedup of 60% in the latest clients for Google Chrome and Firefox.

Other websites with less optimized code could potentially benefit even more. The most used JavaScript library in less optimized websites is JQuery [22] but most websites do not use all its functionality. This type of optimization will benefit even more these websites. We have not done this type of unoptimized websites because they have many other techniques like asynchronous loads and it would be harder to provide a breakdown without understanding each one of them.

Besides the load time optimizations provided by LiveJS, it is a novel solution to improving the load time of webpages. Not only for the techniques deployed but the observation and mechanism to detect functions that are needed for later in the execution time. This observation opens the opportunity to other optimizations not analyzed in this chapter like on-the-fly compilation once the code is sorted by usage.

LiveJS is the first tool that provides the client with the JavaScript functions in a specific order in each file, allowing the web browser to start running the necessary code before the rest of the functions have even begun downloading. LiveJS allows web developers to make no changes when designing and creating webpages, but still allows for a decrease in load time when a website is implemented. As LiveJS is client-independent, it works with multiple web browsers and multiple platforms without any change in performance.

As websites continue to become more complex, the code contained within each website will increase. However, not all of this code is necessary immediately, and thus LiveJS allows for a web developer to ensure that his webpage is provided to the client in a practical way. Load times decrease, as the initial responses for JavaScript files are smaller, therefore allowing the client to start compiling and/or interpreting the code necessary for page loads sooner. Additionally, it parallelizes the requests for JavaScript functionality: the asynchronous request may occur simultaneously with a request for another JavaScript file. That is, instead of having to wait for the previous file to have fully downloaded prior to downloading the next script file, LiveJS will download a shorter version, but still have all the necessary functions.

As LiveJS is completely implemented on the server, future work can include a client-side component. The inclusion of a client component to LiveJS can provide more speedups, by compiling the asynchronously and synchronously downloaded code, eliminating the necessity of the costly `eval()` function call.

Bibliography

[1] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. Flywheel: Google's data compression proxy for the mobile web. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2015)*, 2015.

[2] Mihai Bazon. Uglifyjs. `http://github.com/mishoo/UglifyJS2.git)`, 2016.

[3] Ricardo Bilton. Cnn's latest redesign puts mobile and social at its core, Jan. 2015.

[4] BuiltWith. Javascript usage statitstics, May. 2015. Accessed: 2015-05-23.

[5] Andy Chilton. Javascript minifier, 2015.

[6] Jay Conrod. A tour of v8: full compiler, Dec. 2013.

[7] A Cortesi and M Hils. mitmproxy: a man-in-the-middle proxy, 2016.

[8] Google Developers. Google closure compiler, Feb. 2016.

[9] David Flanagan. *JavaScript: the definitive guide*. "O'Reilly Media, Inc.", 2002.

[10] Xue Fuqiao, Florian Scholz, Karen Scarfone, Berker Peksag, Till Schneidereit, Eric Shepherd, and Chris Leary. Tracing jit, May. 2014.

[11] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for javascript. *SIGPLAN Not.*, 47(6):239–250, Jun. 2012.

[12] Nick Healy. 6 lessons to learn from the new cnn website, Jan. 2015.

[13] Ariya Hidayat. Esprima, Feb. 2016.

[14] Marja Holtta and Daniel Vogelheim. New javascript techniques for rapid page loads, Mar. 2015. Accessed: 2015-05-23.

[15] Bert Hubert. The wonder shaper. `http://github.com/magnific0/wondershaper.git)`, 2012.

[16] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanovic, and Rastislav Bodik. Parallelizing the web browser. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*, 2009.

[17] James Mickens. Silo: Exploiting javascript and dom storage for faster page loads. In *WebApps*, 2010.

[18] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking.

[19] Kailas Patil. Jaegermonkey architecture, Aug. 2011.

[20] Tero Piirainen. Headjs, 2014.

[21] Florian Scholz. Spidermonkey internals, May. 2014.

[22] Steve Souders. Http archive, 2016.

[23] Yusuke Suzuki. escodegen. `https://github.com/estools/escodegen.git`, 2016.

[24] Yusuke Suzuki. estraverse. `https://github.com/estools/estraverse.git`, 2016.

[25] Lucas Vance. Jscompress: Minify javascript online, 2015.

[26] Andy Wingo. v8: a tale of two compilers, Jul. 2011.