DepTracker: Tracking JavaScript Dependencies on the Web

by

Ameesh Kumar Goyal

B.S., Massachusetts Institute of Technology (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

© Ameesh Kumar Goyal, MMXV. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Author

Department of Electrical Engineering and Computer Science

May 22, 2015

Certified by

Hari Balakrishnan
Professor
Thesis Supervisor

Accepted by

Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee
DepTracker: Tracking JavaScript Dependencies on the Web

by

Ameesh Kumar Goyal

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Identifying the performance bottlenecks of Web pages is often the first step in reducing page load times. Existing models of Web pages (dependency graphs) ignore the dynamic interactions of JavaScript objects along these critical paths. Current dependency graphs solely include the dependencies that arise from a Web object triggering a new HTTP request. This thesis presents DepTracker, a tool that captures dynamically generated dependencies between JavaScript objects on a Web page. These JavaScript dependencies give a more accurate picture of the network and computational resources contributing to the critical path.

DepTracker works in conjunction with an HTTP record-and-replay framework, Mahimahi [17], to track reads and writes to the JavaScript global namespace during actual page loads. We classify dependencies into three categories: write-read, read-write, and write-write. Preserving each of these dependencies maintains the consistency of JavaScript execution on Web pages. For each dependency tracked, DepTracker provides developers with relevant line numbers in the source code, variable names, and values that are assigned and read. This information is particularly useful to Web developers seeking to speed up accesses to their websites by reordering individual objects.

We use DepTracker, along with Mahimahi, to expose dependencies on 10 popular Web pages. We find that each Web page includes dependencies between JavaScript objects that are not captured by existing dependency graphs. For our corpus of test sites, we find that graphs that include JavaScript dependencies tracked by DepTracker include 32% and 73% more edges than default dependency graphs, at the median and 95th percentile, respectively.

Thesis Supervisor: Hari Balakrishnan
Title: Professor
Acknowledgments

I first thank my advisor, Prof. Hari Balakrishnan, for his guidance and input over the course of my research. Both during my undergraduate and master’s studies, the opportunity to work with him has been invaluable.

I would also like to thank Ravi Netravali and James Mickens. This project and I have benefited immensely from their collaboration, feedback, and optimism.

I am grateful to the rest of my lab mates in the CSAIL Networks and Mobile Systems Group for their knowledge and contributions to the various projects I have been involved in.

Finally, thank you to my friends and family for their support.
5 Evaluation
   5.1 Results .................................................. 39
   5.2 Case Study: imgur.com .................................. 41

6 Related Work ................................................. 43
   6.1 Modifying the Web page ................................. 43
   6.2 Development Tools ...................................... 44

7 Conclusion .................................................. 47
List of Figures

2-1  Dependency graph of an empty page with a background image set. . . . . . . 19

3-1  Two-file dependency graph without JavaScript dependencies. . . . . . . 22
3-2  Three-file dependency graph without any JavaScript dependencies. . . . 24
3-3  Three-file dependency graph with write-read and write-write dependencies. 25
3-4  Three-file dependency graph with write-read, write-write, and read-write dependencies. ................................. 26

4-1  Dependency graph of an aliased object being modified. ....................... 38

5-1  weather.com’s original dependency graph and new dependency graph with our tracked JavaScript dependencies shown in red. ........................ 40
5-2  outbrain.com’s original dependency graph and new dependency graph with our tracked JavaScript dependencies shown in red. ........................ 40
List of Tables

5.1 The number of edges in the listed Web pages’ dependency graphs grows significantly when our tracked JavaScript dependencies are added. . . . . . 40
Chapter 1

Introduction

Web pages have become increasingly complex in recent years. This complexity is, in part, due to the liberal use of JavaScript. Beyond its original use as a client-side scripting language, JavaScript is now being used to dynamically modify Web pages, often leading to computationally heavy bottlenecks. Recent work [20] found that, on average, 35% of a page’s load time is spent in computation. Moreover, JavaScript is now present in a variety of different settings. For example, browser addons [9], as well as mobile applications [11], are often implemented using JavaScript.

Improving the functionality of applications via JavaScript has a performance cost. JavaScript provides access to many different components of Web pages, including the JavaScript global namespace and the Document Object Model (DOM). Traditionally, the DOM is used as a programming interface for HTML. It provides a structured representation of the Web page as well as functions to modify the page. Ensuring consistency in the JavaScript global namespace and the DOM is difficult because JavaScript is a dynamic scripting language. Thus, static analysis may not capture all relevant dependencies. For example, fetched JavaScript may modify subsequent HTML, preventing browsers from downloading objects further down the page in parallel. Cautiously, modern browsers simply restrict JavaScript processing to be single threaded and blocking with respect to other Web objects (JavaScript, HTML, images, etc.). As a result, much of the parallelism present in modern browsers is negated, adversely affecting page load times.

Users react strongly towards the page load time of websites they visit. Its improvement
has shown significant effects in user experience and business success [4]. In particular, Shopzilla’s revenue increased 10% when its page load time was reduced from 5 to 1.5 seconds [5].

A common approach to overcoming serialization, and reducing page load time, is for developers to reorganize their Web content. Taking advantage of all the parallel network connections browsers can have open can immediately impact the time spent to load the entire page. Additionally, although all JavaScript executes serially in a single thread, deferring the immediate processing of JavaScript files that don’t reside on the critical path can reduce the computational portion of the page load time.

Moreover, the ability to freely reorder content can allow Web developers to optimize for other metrics. For instance, the author of a mobile site may wish to quickly flash the latest headline (using JavaScript) before the rest of the page’s content loads. Similarly, desktop Web applications might choose to dynamically fill in advertisements based on user mouse movement.

The ability to move Web objects freely to optimize such metrics hinges on ensuring consistency between all the objects on a given Web page. Existing structural models of Web pages today (i.e. dependency graphs) solely capture parent-child relationships that arise from a Web object triggering a new HTTP request [20]. Such dependency graphs are too coarse-grained and do not provide developers with enough information to understand the consequences of moving content within their Web pages. Namely, these graphs are missing the dependencies of JavaScript files that happen to be executed earlier but are not necessarily fetched by dependent JavaScript files. Additionally, these dependency graphs are deceiving, as JavaScript objects that contribute to the critical path via computation may not appear to be on the critical path at all!

1.1 DepTracker

To this end, we present DepTracker, a tool that captures dynamically generated dependencies between JavaScript objects on a Web page. DepTracker works in conjunction with Mahimahi [17], an HTTP record-and-replay tool. After recording a Web page with
Mahimahi, DepTracker provides developers with the ability to rewrite the stored Web page. Rewriting is done carefully to ensure that there are no visible changes to Web pages as well as no JavaScript state errors. Pages are rewritten to log reads and writes to JavaScript global variables (objects, functions, etc.). These logs are printed in chronological order of execution during replay as they are directly output from the browser’s JavaScript engine; race conditions are avoided because JavaScript processing is single threaded in modern browsers.

DepTracker includes a log processor that converts logs into parent-child relationships (dependencies) between JavaScript Web objects. We classify dependencies into three groups:

1. Write-Read dependencies: \texttt{b.js} depends on \texttt{a.js} if \texttt{b.js} reads a variable whose value was written in \texttt{a.js} and no other script modifies the variable between the execution of \texttt{a.js} and \texttt{b.js}.

2. Read-Write dependencies: If \texttt{a.js} reads a variable which was written sometime earlier, and \texttt{b.js} later writes the same variable, then \texttt{b.js} must not come before \texttt{a.js}.

3. Write-Write dependencies: If \texttt{a.js} writes a variable, \texttt{b.js} subsequently writes the same variable, and no other scripts write this variable, then \texttt{a.js} must come before \texttt{b.js} to preserve the end state of the page.

To better assist Web developers in restructuring their Web pages, each dependency output by the log processor is annotated with useful debugging information. This includes the relevant line numbers in both the parent and child object’s source code, variable names, assigned and read values, etc.

We use DepTracker, along with Mahimahi, to expose dependencies on several Web pages in the Alexa US Top 500 [3]. We find that each Web page includes dependencies between JavaScript objects beyond those captured by default dependency graphs. Moreover, we find that graphs that include JavaScript dependencies tracked with DepTracker include 32% and 73% more edges than default dependency graphs, at the median and 95th percentile.
Chapter 2

Background

We first describe how browsers load Web pages. To request a Web page, the browser issues an HTTP GET to the root path (/) of the relevant Web page address. The Web server will respond with an HTML index file that contains references to various other resources. These may include JavaScript files, CSS files, images, more HTML files, etc.

As the browser parses the response of the / request, it builds up a list of requests that it must make and downloads them. Concurrently, as responses for requests are received, objects are evaluated and rendered onto the page. The browser is subject to constraints in how many network requests it may have open at any given time. Currently, the limitation for Firefox is 6 requests per origin and 256 requests overall [15][16].

It is commonly the case that some embedded objects of the root-level index file make requests of their own. For instance, a CSS file may reference an image to be downloaded and set as the background of the page. Additionally, JavaScript files may send AJAX requests for other JavaScript files.

2.1 Dependency Graphs

The requests a browser makes and the relevant requesting objects can be represented as a directed graph of object (i.e file) dependencies. This dependency graph contains a node for every object that the browser downloads. A directed edge exists from each object to each object that file requested. For example, consider the index file below and the simple CSS
file it references:

index.html
<html>
<head>
    <link rel="stylesheet" type="text/css" href="foo.css">
</head>
<body>
</body>
</html>

foo.css
body {
    background-image: url("background.jpg");
}

Loading this Web page in the browser will result in an empty page with the background displaying background.jpg. To achieve this result, the browser will first send an HTTP GET request for the root-level HTML file index.html. Upon parsing index.html, the browser will notice that it must make a request for foo.css. Subsequently, upon parsing foo.css, the browser will request background.jpg.

The dependency graph, as described above, for this sample Web page is shown in Figure 2-1. It is clear from this graph that the background.jpg file depends on execution of the foo.css file, which in turn depends on execution of the root / file.

Certain file types can only appear in particular areas of this graph. For instance, images cannot initiate requests for other objects, so images are always leafs. CSS, JavaScript, and HTML files (which may appear as part of an iframe node), can initiate requests for zero or more other objects. Thus, they can appear as parents or leafs in the graph.

Dependency graphs clearly display key metrics of Web pages. For example, the number of nodes is the number of requests the browser had to make, excluding caching, to download the Web page. Additionally, the maximum depth of the graph is the minimum number of
round trip times (RTTs) to the Web server the browser would have to make to download the page. This max depth is commonly referred to as the critical path. Due to browser constraints mentioned earlier, it may take even more RTTs to actually download the entire page.

The edges in dependency graphs can be used to determine which files are contributing the most to this critical path. It is in a Web developer’s best interest to minimize the length of the critical path to maintain a fast Web page. Changes in a Web page’s dependency graph can significantly affect the page’s load time.

It must be noted, however, that the dependency graph as described above does not contain all the dependencies present on a Web page. §3 describes the JavaScript dependencies that may be missing, and §4 describes the algorithm DepTracker employs to find these dependencies automatically.
Chapter 3

JavaScript Dependencies

3.1 Global Variables

JavaScript files in a Web page may exhibit dependencies on one another through shared global state. For example, consider the following JavaScript files and the index file that includes them.

a.js
x = 1;

b.js
console.log(x);

index.html
<html>
  <head>
    <script src="a.js"/>
    <script src="b.js"/>
  </head>
</html>
In the serialized order a→b, the value 1 will be printed to the browser console. However, if b is executed before a, a reference error will be thrown because x is not yet defined. Therefore, a JavaScript dependency exists between file a and file b because of the shared global variable, x. These JavaScript dependencies are not captured by simply observing URL fetches as described in §2. In fact, the dependency graph as described in §2 will have no dependencies between files a and b at all. This original graph is shown in Figure 3-1 for reference.

When considering the parent-child relationship of these JavaScript dependencies, we assume a "write" is a parent of a "read". In this case, file a’s write to x and file b’s subsequent read of x would create an a→b dependency in the dependency graph shown in Figure 3-1.

### 3.2 Global Functions

The previous example illustrates a shared global variable dependency. JavaScript functions can also be defined globally, as shown below.

```javascript
// a.js
function y() {
    console.log("foo");
}
```
In this example, \(a\) defines function \(y\) at the global scope, and \(b\) calls function \(y\). Therefore, a JavaScript dependency exists between \(a\) and \(b\) because of the global function, \(y\). Because \(a\) defines the function \(y\), \(a\) would be a parent of \(b\) in this dependency.

### 3.3 Global Objects

Moreover, JavaScript objects may be passed between files and possibly modified, again creating a dependency between the relevant files. Consider the example below:

**a.js**
```javascript
foo = {};
foo.bar = {};
foo.bar.a = 1;
```

**b.js**
```javascript
foo.bar.a = 2;
```

Here, \(a\) defines a global object \(foo\), and alters some nested properties of the object. \(b\) then again alters a nested property of \(foo\), namely the \(bar.a\) property, but does not modify the global object \(foo\) itself. Although the top-level object is not modified, executing \(b\) before \(a\) will result in a reference error. This suggests that there is a JavaScript dependency from \(a\) to \(b\) due to the write of \(foo.bar.a\) and the subsequent write of \(foo.bar.a\) in \(b\).

### 3.4 Dependency Types

This last example illustrates a write-write dependency, in contrast to the previous examples of write-read dependencies. In order for our dependency graph to reflect the same order of serialization as a normal browser page load, we represent write-write dependencies as
Figure 3-2: Three-file dependency graph without any JavaScript dependencies.

parent-child relationships where the parent is the earlier write and the child is the later write.

Capturing write-write dependencies and write-read dependencies will ensure that the order of writes to an object is preserved and reflected in the resulting dependency graph. However, these dependencies are not sufficient to ensure all dependencies between JavaScript files are accounted for. For instance, consider the following three JavaScript file example:

```
a.js
foo=1;

b.js
console.log(foo);

c.js
foo=2;
```

The dependency graph for this example, with solely network dependencies as described in §2, is shown in Figure 3-2. If the actual browser execution order is a, b, c, the dependencies defined above would create a write-read dependency from a to b due to the write of foo in a and the read of foo in b. There would also be a write-write dependency from a to c due to the write of foo in a and the write of foo in c. The resulting graph is shown in Figure 3-3:
This indicates that there is no dependency between \( b \) and \( c \). However, if \( c \) is executed before \( b \), the output of the \( \text{console.log}(\text{foo}); \) line will be different. This suggests that there should be a dependency between \( b \) and \( c \). To this end, we track read-write dependencies. There is a read-write dependency from \( b \) to \( c \) due to the read of \( \text{foo} \) in \( b \) and the write of \( \text{foo} \) in \( c \). Given this new dependency, we have the complete dependency graph shown in Figure 3-4.

JavaScript files on a Web page can introduce dependencies on one another through shared global state, as described above. There are a variety of methods to introduce dependencies, including shared global functions and global objects. To track these dependencies, we classify each dependency into one of three types:

- write-read dependency
- write-write dependency
- read-write dependency

Each dependency type defines a parent-child relationship. When added to the dependency graph described in §2, these new relationships result in a complete set of dependencies on a Web page. When all dependencies between objects on a Web page are known, developers can be cognizant of the consequences of reordering objects within the page.
Figure 3-4: Three-file dependency graph with write-read, write-write, and read-write dependencies.
Chapter 4

Design

We show here how DepTracker automatically tracks the JavaScript dependencies detailed in §3. We describe a method for logging reads and writes to global objects and functions in JavaScript, and then extract the parent-child relationships between JavaScript files in post-processing.

As explained in §3, global state that is shared across JavaScript files results in dependencies between JavaScript files. Global state in JavaScript is determined by the scope in which the state, i.e. the variable, is defined. Note that, in JavaScript, objects and functions are also variables.

4.1 JavaScript Scoping

JavaScript is a functionally scoped language. All local variables are only recognized within the local function they are defined in and the nested functions therein. Variables declared outside of any functions are in the global scope, and can be accessed by other JavaScript files on the Web page. Moreover, variables declared within a function default to becoming global variables. A variable must be explicitly declared using the `var` keyword in order to become a local variable.

All global variables in JavaScript, including global objects and global functions, are automatically properties of the global `window` object. For example, a global variable declared as `foo=1;` can be subsequently accessed as `window.foo` or `foo`, interchangably.
This applies to functions declared in the global scope as well.

The analysis in §3 showed that global variables can lead to dependencies between JavaScript files. We describe above that all global variables in JavaScript are properties of the window object. Using these two facts together, we create DepTracker, a tool for tracking dependencies between JavaScript files.

MDN’s experimental Proxy object can be used to track accesses to the window object, and therefore all global variables [18]. Using these accesses, we can apply post processing to get the actual parent-child relationships between JavaScript files.

4.2 Proxy Objects

MDN’s Proxy object allows a user to specify handlers for fundamental operations to a target object. For our purposes, the target object is window. We define traps for the get and set handlers to the window object. Inside each handler, we log a READ or WRITE (for get or set, respectively) for the applicable property accessed in window. For example, a read of a global variable window.foo would hit our get trap, where we would log a READ to foo with the reading file’s name.

One caveat to the window Proxy just described is that global variables only hit our trap if they are explicitly accessed as properties of window. Continuing the previous example, although window.foo and foo reference identical variables, only the use of window.foo would hit our Proxy. This introduces the problem of statically rewriting JavaScript files to prepend window. to all global variables.

To recap, we have just described a method of tracking global variables, which are properties of the window object, across files. By inserting a Proxy between window gets and sets, we can log what global variables are accessed when, and by which file. However, in order for the Proxy to function properly, we must ensure that all global variables are referred to as properties of the window object.
4.3 Record and Replay

In order to prepend `window` to all global variables, we must have a method of recording Web pages, modifying them, and replaying them to the browser. Conveniently, an HTTP record-and-replay tool, Mahimahi [17], provides the framework we need.

DepTracker records Web pages that need to be analyzed using Mahimahi’s recordshell, and then applies the `window` rewriting algorithm described in §4.4 to the recorded content. Once the Web page content has been modified appropriately, DepTracker opens Mozilla Firefox inside of a Mahimahi replayshell for that Web page. Navigating to that Web page in Firefox replays the modified recorded content. The captured log output of this page load can be later processed using DepTracker’s Log Processing tool (§4.6) to extract the actual parent-child dependencies.

Currently, Mozilla Firefox is the only browser supporting the MDN Proxy object. Unfortunately, Firefox does not provide a way to automatically disable HTTPS certificate errors, which, of course, are present when replaying content locally using Mahimahi. Therefore, until Firefox is able to suppress HTTPS certificate errors, or another universal browser (such as Google Chrome) provides support for the Proxy object, DepTracker is limited to sites that operate solely on HTTP. We do not anticipate this becoming a major design flaw of DepTracker, as Proxy objects are under active development for Google Chrome [7].

4.4 Window Rewriting Algorithm

We employ the Esprima parser to create an abstract syntax tree (AST) out of the JavaScript code we want to statically rewrite [10]. Estraverse is used to traverse this AST and update the relevant global variables to be prepended with `window` [19]. We accomplish this in three traversals through the tree.

To illustrate why this type of `window` rewriting cannot be accomplished with just one traversal through the AST, we can examine some of the intricacies of JavaScript. Consider the following JavaScript function:

```javascript
function foobar() {
  
```
It appears as if foobar sets global variable a to be 1, sets global variable b to be 2, and then defines a local variable a to be 8. However, in the first line of the function, the variable a is not global. var declarations in JavaScript are hoisted to the top of functions by a JavaScript preprocessor, prior to execution of the code. In this case, the first line of the function actually refers to a local variable a. It is not clear, when traversing an AST of foobar, that the first reference to a is a reference to a local variable. We would have to traverse to the last line of the function before knowing that the variable had been declared as a local var. Thus, rewriting all global variables in one traversal through the AST is not possible unless considerable state is maintained for all variables in the entire JavaScript file.

Our three-step traversal algorithm consists of the following steps:

1. Catalog all variables defined in the file and the scope they are defined at.
2. Rewrite all global variables to be prepended with window.
3. Hoist all rewritten function expressions to the top of the file, akin to var declaration hoisting by the JavaScript preprocessor.

4.4.1 Step 1

In the first step, we traverse through the AST and maintain a mapping between the current scope we are in and the local variables declared at that scope. The current scope is maintained as a comma-separated string of function names, ordered from outermost to innermost function. In the case of anonymous functions, we name the i\(^{th}\) anonymous function "anon" concatenated with i, and maintain the counter i across the entire file. For example, consider the following JavaScript code:
a = 1;
function b() {
    function c() {}
    d = 1;
    e = 2;
    a = 3;
    var d;
}

After executing the first step of our algorithm, the resulting mapping would be:

["global"] -> ["a", "b"]
["global,b"] -> ["c", "d"]
["global,b,c"] ->[]

Notice that variable e is missing from the list of declared local variables due to the fact that it was not preceded by a var keyword. Also, the d variable is always referenced locally due to the var d; declaration being hoisted to the top of function c.

4.4.2 Step 2

Using this mapping, we proceed with the second step of our algorithm - a second pass through the AST. On this second pass, we examine all variable references and declarations. If a variable is not declared as a local variable in the current scope, or any scope containing it (excluding the outermost global scope), then it is a global variable. Continuing with the example code above, the e variable is not locally declared in any scope containing it, and is therefore a global variable.

Once it is determined that a variable declaration or reference is global, we can safely rewrite it to be prepended with window. Additionally, we can add an _id property to any objects, which is described in more detail in §4.6. There may seem to be some concern with modifying the AST while iterating through it. However, this is not an issue because the modifications to the AST are made only to the current node and do not affect the previous
or upcoming nodes in the AST. In our example above, the declaration of `e=2;` would be rewritten to `window.e=2;`.

In addition to rewriting variables that do not appear in any scope containing them, we rewrite all variables at the outermost global scope. This is because all variables defined outside of any functions are automatically global. For our example above, this would include rewriting the declaration of `a=1;` to `window.a=1;`, as well as rewriting the function `b` to be prepended with `window`. We must note that only functions declared in the global scope will need to be rewritten in this manner because any nested functions cannot be global by the definition of JavaScript scoping.

The rewriting of function declarations, including function `b` in the example above, poses a problem. JavaScript does not allow function declarations to contain punctuation, let alone be assigned as properties of other objects. To address this issue, we can rewrite function declarations into anonymous function expressions, where we can assign the function as a property of `window` immediately. For example, the function `a` below could be rewritten as follows:

```javascript
function a() {
    sample = 1;
}
```

//rewritten to
```javascript
window.a = function () {
    sample = 1;
}
```

Function expressions and function declarations are identical to the caller, but the benefit with function expressions is that the function identifier can be the property of another object. Here, the rewritten function `a` will now hit the `get` handler in our `window` Proxy.

However, function expressions and function declarations are treated differently by the JavaScript preprocessor. Function declarations behave the same as `var` declarations, in that they are hoisted to the top of the containing function by the preprocessor. Function
expressions are not hoisted. This can pose a problem when rewriting, for example, the following code:

```javascript
foo = bar();
function bar() {
    return 1;
}
```

//rewritten to
```javascript
window.foo = window.bar();
window.bar = function() {
    return 1;
}
```

In the original code on the top, a global variable `foo` is assigned the output of the global function `bar`, which is defined on the next line. This does not pose a problem because the function declaration of `bar` will be hoisted to the top of the file by the JavaScript preprocessor, allowing the call to `bar` on the first line to reference the correct function.

In the rewritten code, all the global references are prepended with `window`, and the `bar` function declaration is rewritten as a function expression. Upon execution, the call to `window.bar()` will result in a reference error. This is because the function `window.bar` is not defined until the next line, and the JavaScript preprocessor will not hoist it to the top of the file before execution.

### 4.4.3 Step 3

To address this type of issue, we do not rewrite function declarations to function expressions in the way described above. We leave function declarations as they were and instead apply the third and final step of our algorithm involving a third pass through the AST. In this final pass, we mimic a JavaScript preprocessor and hoist function expressions at the global scope to the top of the file. To continue with the example above, now the code would be rewritten to:
window.bar = bar;
window.foo = window.bar();
function bar() {
    return 1;
}

The rewritten code rewrites all global references to be prepended with `window`. It does not modify the function declaration at all. Additionally, we prepend a line (`window.bar = bar;`) to the top of the file. Because the function declaration will be hoisted to the top of the file by the JavaScript preprocessor, `bar` will be defined on this first line upon program execution. Prepending this line to file will log a WRITE to the `window.bar` variable, and then the next line containing the call to `window.bar()` will log a READ. This is the behavior we wished to achieve using the erroneous function declaration rewriting method described previously. With this method, we do not run into any unexpected missing reference errors.

This last pass through the AST is the final step in our rewriting algorithm. After this step, all global variables (i.e. variables, functions, and objects) will be prepended with `window.`, and will hit our `window` Proxy. There a few edge cases which are detailed in the tracking code itself. A particularly interesting edge case, concerning tracking global objects across files, is discussed in §4.5.

### 4.5 Recursive Proxies

All JavaScript objects are associative arrays, also referred to as hashes. The keys in these objects are strings, and values can be primitive types or other objects. Tracking accesses to nested objects can pose some interesting problems. For example, consider the rewritten code below:

```javascript
a.js
window.foo = {};
window.foo.bar = {};
```
window.foo.bar.zoo = 1;

b.js
window.foo.bar.zoo = 2;

The global foo object after execution of a will become:

{"bar": {"zoo": 1}}

Then executing b will modify the foo object to:

{"bar": {"zoo": 2}}

After executing this example, we would like to extract the write-write dependency from a to b due to the write of foo.bar.zoo. However, our Proxy will only track changes to the top level object because only the top level object is a property of window. All nested objects are just references from the top level object. In this case, our Proxy would track only the read of the bar object of foo and would result in only a write-read dependency from a.js to b.js, which is incorrect.

To address this issue, we introduce recursive proxies. When returning from our window Proxy, if the returned value is an object, we return a Proxy whose target is set to that object. When the returned object is next accessed, it will be caught by that Proxy. In the example above, we can follow the steps of tracking the write in b.

foo is a property of window, which is of course a Proxy. Upon returning from the window Proxy, we notice that foo is itself an object, and return a Proxy wrapped on that foo object. When the JavaScript engine tries to access the bar property of the foo object, the same steps occur. Then, when the zoo property of the bar object is accessed, the set trap in the Proxy on bar will note that the zoo property is a primitive (namely, an integer). We do not make a Proxy for the zoo property, and will instead log a WRITE to the zoo property in this file.

Recursive proxies address the issue of only logging changes and accesses to the top level object. By applying proxies to all objects, we gain the ability to track changes to arbitrarily nested objects.
4.6 Log Processing

All of our proxies log WRITEs or READs of variables to the console. In post processing, we can match WRITEs or READs of the same variable across files, determine which type of dependency it is as described in §3.4, and then extract the actual parent-child dependencies between files.

Each log entry that we process contains the relevant filename, the line number in that file, whether it was a WRITE or READ, the name of the variable, the value assigned or read, and, if the variable is an object, the _id of the object. The _id of the object is added in Step 2 of our algorithm (§4.4.2) and is useful when tracking changes to an object referenced by multiple variables. For example, consider the following three JavaScript files:

a.js
foo = {};

b.js
bar = foo;

c.js
bar.a = 1;

If executed in the order a-b-c, foo and bar will point to the same object which has a value of {"a": 1}. Our proxies would log a WRITE to foo in a, a READ of foo and a WRITE to bar in b, and a WRITE to bar.a in c. This would result in a becoming a parent of b and b becoming a parent of c. However, there would be no direct dependency between a and c, even though foo and bar point to the same object.

To handle these types of aliasing issues, we extend all objects to have an _id property. The _id is simply a counter uniquely identifying each object. When processing the log, we can match READs and WRITEs to objects by _id instead of variable name.

To extract the parent-child dependencies, we apply a brute force algorithm. Consider the logs below generated for the previous example.
Each log entry is formatted with the type of access, the variable name, the file name, the object _id, the value assigned or read, and then the line number in the relevant file. Because JavaScript is a single-threaded language, the log entries appear in chronological order.

When an _id is present, we do all variable matching based on that _id. In this case, all of the entries refer to the same object _id. As we iterate through the log entries, we take note of any READs and iterate backwards through the log to find the relevant WRITE. This will extract write-read dependencies. Subsequently, for each READ, we iterate down the log and find the next WRITE to the same variable. This extracts read-write dependencies.

To find write-write dependencies, we iterate through the log backwards starting from the bottom. We introduce a write-write dependency from every file with a WRITE to a variable to the file with the last WRITE to that variable. Including this dependency ensures that the end state of every variable persists.

Including all write-read, read-write, and write-write dependencies may result in many duplicate dependencies between files. We apply a de-duplication step to remove this redundancy, and then are left with the parent-child dependencies between files. In the example log above, these dependencies would produce the dependency graph in Figure 4-1.
Figure 4-1: Dependency graph of an aliased object being modified.
Chapter 5

Evaluation

We use DepTracker, along with Mahimahi, to expose dependencies on several Web pages in the Alexa US Top 500 [3]. As discussed in §4.3, DepTracker is currently limited to Web pages that do not employ HTTPS. In the near future, once wider support for the MDN Proxy object is garnered, DepTracker will support HTTPS pages as well.

We find that each Web page includes dependencies between JavaScript objects beyond those captured by default dependency graphs. Moreover, we find that dependency graphs that include JavaScript dependencies tracked with DepTracker include 32% and 73% more edges than default dependency graphs, at the median and 95th percentile, respectively.

5.1 Results

Table 5.1 shows the growth in the number of edges in the dependency graph for 10 of the Top 500 sites. Note that the number of objects, i.e. nodes, in the dependency graph does not change. This is due to the fact that DepTracker preserves all the content on Web pages and does not introduce any new objects.

In some cases, DepTracker adds a large number of new dependencies to the graph. For example, Table 5.1 shows that DepTracker found 59 new dependencies in weather.com’s Web page. Figure 5-1a depicts weather.com’s original dependency graph, and Figure 5-1b shows the same graph with the new edges in red. The added JavaScript dependencies make the Web page’s complexity much easier to see.
Table 5.1: The number of edges in the listed Web pages’ dependency graphs grows significantly when our tracked JavaScript dependencies are added.

<table>
<thead>
<tr>
<th>Web page</th>
<th># of nodes</th>
<th>original # of edges</th>
<th># of edges with new dependencies</th>
<th>% increase in # of edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple.com</td>
<td>28</td>
<td>27</td>
<td>36</td>
<td>33%</td>
</tr>
<tr>
<td>ask.com</td>
<td>65</td>
<td>72</td>
<td>94</td>
<td>31%</td>
</tr>
<tr>
<td>bing.com</td>
<td>30</td>
<td>29</td>
<td>46</td>
<td>59%</td>
</tr>
<tr>
<td>cbssports.com</td>
<td>107</td>
<td>108</td>
<td>188</td>
<td>74%</td>
</tr>
<tr>
<td>imdb.com</td>
<td>108</td>
<td>118</td>
<td>135</td>
<td>14%</td>
</tr>
<tr>
<td>imgur.com</td>
<td>102</td>
<td>101</td>
<td>129</td>
<td>28%</td>
</tr>
<tr>
<td>m.finishline.com</td>
<td>111</td>
<td>114</td>
<td>137</td>
<td>20%</td>
</tr>
<tr>
<td>outbrain.com</td>
<td>81</td>
<td>82</td>
<td>131</td>
<td>60%</td>
</tr>
<tr>
<td>stackoverflow.com</td>
<td>37</td>
<td>36</td>
<td>43</td>
<td>19%</td>
</tr>
<tr>
<td>weather.com</td>
<td>82</td>
<td>82</td>
<td>141</td>
<td>72%</td>
</tr>
</tbody>
</table>

Figure 5-1: weather.com’s original dependency graph and new dependency graph with our tracked JavaScript dependencies shown in red.

Figure 5-2: outbrain.com’s original dependency graph and new dependency graph with our tracked JavaScript dependencies shown in red.
DepTracker achieves a similar result in the case of outbrain.com. Table 5.1 shows that DepTracker found 49 new JavaScript dependencies, which are graphed in red in Figure 5-2b. In this case, fewer files are contributing to the added dependencies than in weather.com, but the dramatic increase in depth and complexity is again apparent.

5.2 Case Study: imgur.com

It is interesting to inspect which particular lines of code contribute to the added JavaScript dependencies seen on the Web. Because DepTracker provides the line numbers and variable names in each log entry (as described in §4.6), we can simply lookup the applicable lines of code. For example, a new dependency found in imgur.com (also displayed in Table 5.1) occurs between the global.js and index.js files on its Web page. The lines of code causing this dependency are shown below:

//excerpt of global.js
window.Namespace = function () {
    var a, b, c, d = arguments, e = null;
    for (a = 0; a < d.length; a += 1)
        e = window;
    for (c = d[a].split('.'), b = 0; b < c.length; b += 1)
        e[c[b]] = e[c[b]] || makeProxy({}), e = e[c[b]];
    return e;
}

//excerpt of index.js
window.Namespace('Imgur.Upload.Index');

global.js defines a global function called Namespace, which index.js later invokes. This is a write-read dependency due to the global Namespace function, which explains the new dependency between these files on imgur.com.
Chapter 6

Related Work

There has been much work done aimed at decreasing Web page load time. Among the techniques to improve this metric are systems that modify the transport layer between the Web page and the Web server, and systems that modify the Web page itself. DepTracker falls into the latter category, although it is complementary to many other techniques that reduce page load time. For instance, DepTracker could benefit from recent transport layer improvements like Google’s QUIC [1] and SPDY [2].

6.1 Modifying the Web page

Google’s mod_pagespeed [8] provides an Apache module that automatically applies Web performance best practices to reduce page load time. However, the tool does not provide fine-grained control for selectively loading objects in a defined order. It simply supplies generic tools that can be applied to a wide variety of Web pages with minimal effort on the part of the Web developer, e.g. minifying CSS and JavaScript. Moreover, mod_pagespeed is agnostic to the structure of the Web page’s dependency graph. A study [20] on its performance found that mod_pagespeed has little impact on the page load time because most of the time spent on the critical path is in network fetches.

Silo [14] takes the approach of inlining JavaScript and CSS files to minimize HTTP fetches. This approach is not fruitful when considering objects which are not immediate children of the root HTML file. Many objects may exist past this depth in the dependency
graph. As discussed in §2.1, it may be the depth of this graph that contributes most to the page load time. DepTracker provides the developer insight on improving page load time using all Web objects, regardless of where they occur in the dependency graph. In fact, Silo’s benefits for the immediate dependencies of the root HTML file can be compounded with information DepTracker provides to further reduce page load time.

6.2 Development Tools

Related tools that aid Web developers in improving page load time are not nearly as complete or fine-grained as DepTracker. WProf [20] is a developer tool most similar to DepTracker, but it ignores any JavaScript dependencies. The dependency graph WProf uses in its analysis is based on HTTP fetches, which, as shown in §5.1, can be a significant underestimate of the dependencies actually present on a Web page. Moving resources based only on information gained from HTTP fetches is inherently error-prone because JavaScript files do not necessarily have to be fetched by dependent JavaScript files. A case in point is jQuery [12], which is commonly fetched once and used by multiple other scripts.

WProf borrows from the work done on WebProphet [13]. WebProphet aims to extract object dependencies on Web pages by perturbing the network connection for each Web object. This is an orthogonal technique to obtaining dependencies via solely HTTP fetch initiators. However, WebProphet does not include computation time in its dependency calculations, so its resulting dependency graphs are inherently incomplete.

Google PageSpeed Insights [6] provides information about the critical path for a specific page load. However, it does not disclose any of the other dependencies on the page and is likely inaccurate when objects in the page load are cached. Moreover, PageSpeed is a closed-source resource provided by Google, so there is no opportunity to control the network conditions during the page load itself. DepTracker employs Mahimahi, which allows users to specify the network conditions and test how the critical path changes.

In-browser development tools, like Firebug [9], provide waterfall timing graphs for all the objects fetched via the network, as well as the initiators that fetched those objects. This list of objects and initiators can be used to construct a dependency graph, but, again, this
graph would be based solely on HTTP fetches. Additionally, Firebug does not construct this graph automatically. Inferring the structure of this graph and searching for the critical path via a list of nodes and edges can be cumbersome.
Chapter 7

Conclusion

Users are easily frustrated with slow Web pages. Significant usability and even revenue gains can be attained with improved Web page load times. Consequently, a great deal of prior work has aimed towards this effect. A common technique to reduce page load time involves removing Web objects from the bottleneck path of network fetches and execution. This technique is attractive because it is complementary to much of the other work done in the area, such as improvements to transport layer protocols and minifying JavaScript.

Reordering individual objects is difficult without fine-grained detail of how these objects interact with each other. Existing dependency graphs are incomplete; they are missing the dependencies that JavaScript objects impose on one another. Without this knowledge, maintaining the consistency of JavaScript execution when moving objects around becomes a tedious trial-and-error process.

To this end, this thesis introduces DepTracker, a tool that captures these missing, dynamically generated dependencies between JavaScript objects. DepTracker employs the HTTP record-and-replay framework, Mahimahi, to track these dependencies during actual page loads. The line numbers and variable names that DepTracker provides for each dependency are particularly useful when reordering objects on the page.

In our own tests, we find that graphs that include JavaScript dependencies tracked by DepTracker include 32% and 73% more edges than default dependency graphs, at the median and 95th percentile, respectively.
Bibliography


