Doloto: Code Splitting for Network-Bound Web 2.0 Applications

Benjamin Livshits
Microsoft Research

Emre Kıcıman
Microsoft Research

Abstract

Modern Web 2.0 applications, such as GMail, Live Maps, Facebook and many others, use a combination of Dynamic HTML, JavaScript and other Web browser technologies commonly referred as AJAX to push page generation and content manipulation to the client web browser. This approach improves the responsiveness of these network-bound applications, but the shift of application execution from a back-end server to the client also often dramatically increases the amount of code that must first be downloaded to the browser. This creates an unfortunate Catch-22: to create responsive distributed Web 2.0 applications developers move code to the client, but for an application to be responsive, the code must first be transferred there, which takes time.

In this paper, we present DOLOTO, a system that analyzes application workloads and *automatically* performs code splitting of existing large Web 2.0 applications. After being processed by DOLOTO, an application will initially transfer only the portion of code necessary for application initialization. The rest of the application's code is replaced by short stubs—their actual function code is transfered lazily in the background or, at the latest, on-demand on first execution. Since code download is interleaved with application execution, users can start interacting with the Web application much sooner, without waiting for the code that implements extra, unused features.

To demonstrate the effectiveness of DOLOTO in practice, we have performed experiments on five large widely-used Web 2.0 applications. DOLOTO reduces the size of initial application code download by hundreds of kilobytes or as much as 50% of the original download size. The time to download and begin interacting with large applications is reduced by 20-40% depending on the application and wide-area network conditions.

1 Introduction

Over the last several years, we have seen the creation of a new generation of sophisticated distributed Web 2.0 applications as diverse as GMail, Live Maps, RedFin, MySpace, and NetFlix. A key enabler for these applications is their use of client-side code—usually JavaScript executed within the Web browser—to provide a smooth and performant experience for users while the rendered Web page is dynamically updated

in response to user actions and client-server interactions. As the sophistication and feature sets of these Web applications grow, however, downloading their client-side code is increasingly becoming a bottleneck in both their initial startup time and subsequent application interactions. Given the importance of performance and "instant gratification" in the adoption of today's applications, a key challenge thus lies in maintaining and improving application responsiveness despite increasing code size.

While much work has been done on improving server-side Web application performance and reducing the processing latency [13, 19, 20], recent studies of modern Web 2.0 applications indicate that front-end execution contributes 95% of execution time with an empty browser cache and 88% with a full browser cache [17]. Moreover, browser-side caching of Web content is less effective than previously believed because about 40% of users come with an empty cache [16]. This, along with a trend towards network delivery of increasingly sophisticated distributed Web applications, as exemplified by technologies such as Silvelight, highlights the importance of client-side optimizations for archieving good end-to-end application performance.

Indeed, for many of today's popular Web 2.0 applications, client-side components are already approaching or exceeding 1 MB of code (uncompressed). Clearly, however, having the user wait until the *entire* code base is transferred to the client before the execution can commence does not result in the most responsive user experience, especially on slower connections. For example, over a typical 802.11b wireless connection, the simple act of opening an email in a Hotmail inbox can take 24 seconds on a first visit. Even on a second visit takes 11 seconds—even after much of the static resources and code have been cached. Users on dial-up, cellphone, or slow international networks see much worse latencies, of course, and large applications become virtually unusable. Live Maps, for example, takes over 3 minutes to download on a second (cached) visit over a 56k modem. (According to a recent Pew research poll, 23% of people who use Internet at home rely on dial-up connections [14].) In addition to increased application responsiveness, reducing the amount of code needed for the application to run has the benefit of reducing the overall download size which is important in the mobile and some international contexts, where the cost of network connectivity is often measured per byte instead of a flat rate approach common in North America.

One solution is to structure Web application code such that only the minimal amount of code necessary for initialization is transferred in the critical path of Web application loading; the rest of the application's code would be dynamically loaded as bandwidth becomes available or on-demand. While modern browsers do support explicit loading of JavaScript code on-demand, after a Web page's initial download, few applications make extensive use of this capability. The issue is that manually architecting a Web application to correctly support dynamic loading of application code is a challenging and error-prone process. Web developers have to track the dependencies between user actions, application functionality, and code components. They have to schedule background downloads of code at the appropriate times to avoid stalling a user's interactions. Finally, developers have to maintain and update the resultant code base as the application code and typical user workloads evolve.

1.1 Overview of Doloto

In this paper, we propose DOLOTO[1], a tool that performs automated analysis of Web application workloads and automatic code splitting as a means to improve the responsiveness of Web 2.0 applications and reduce their download size.

Code splitting in DOLOTO is performed at the level of individual functions, which are clustered together to form an *access profile* computed based on function access times computed during a *training phase* collected with the help of runtime instrumentation. In the *execution phase*, access profiles guide the process of on-demand code loading so that functions with similar access times are clustered to be downloaded together. Additionally, whenever network bandwidth becomes available, code is transferred to the client via a background prefetching queue. As part of DOLOTO rewriting, function definitions are replaced with short stubs that block to fetch actual function bodies whenever necessary. Doing so in a sound manner is quite tricky in a language such as JavaScript that supports high-order functions and `eval`. In particular, to preserve the scoping of function definitions in the original program, we still eagerly transfer function *declarations*, however, function bodies are transferred lazily. DOLOTO effectively introduces dynamic code loading to applications that have been developed without it in mind.

To show the effectiveness of DOLOTO in practice, we have performed an evaluation on a set of five large widely-used Web 2.0 applications for a range of of bandwidth and latency values. The benefits of code splitting become especially pronounced for slow connections, where the initial page loading penalty is especially high if transferring the entire code base. DOLOTO reduces the size of initial application code download by hundreds of kilobytes or as much as 50% of the original download size. The time to download and begin interacting with large applications is reduced by 20-40% depending on the application and wide-area network conditions. Moreover, with background code loading enabled, the rest of the application can be downloaded *while the user is interacting with the application*. In our experiments it took 30-63% extra time compared to the original application initialization time for background prefetch to download the entire application code base.

1.2 Contributions

This paper makes the following contributions:

- We propose code splitting as a means of improving the perceived responsiveness of large and complex distributed Web 2.0 applications within the end-user's browser.
- We describe a code rewriting strategy that breaks the code of JavaScript functions into small stubs that are transferred to the client eagerly and fetches remaining function bodies at runtime on-demand, without requiring application-specific knowledge or changes to existing code. We develop a runtime instrumentation approach that, together with a clustering scheme to analyze the data gathered at

[1] *Doloto* is Russian for *chisel*.

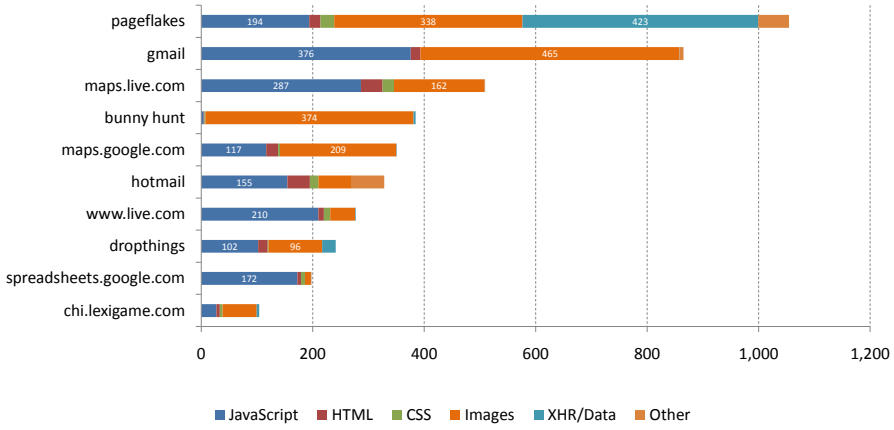

Figure 1: Size breakdown of different Web 2.0 application components. Provided sizes are after gzip compression.

training time, helps us cluster portions of client-side JavaScript code that exhibit temporal locality.

- We perform a detailed evaluation of DOLOTO for a set of five popular Web 2.0 applications and demonstrate the effectiveness of our techniques for a range of network conditions.

1.3 Paper Organization

The rest of the paper is organized as follows. Section 2 provides background on Web 2.0 applications and also gives an overview of common application construction patterns, motivating why code splitting is a good strategy for improving application responsiveness. Section 3 gives a description of our training and code rewriting techniques. Section 4 discusses our experimental results. Finally, Sections 5, 6, and 7 describe related and future work and provide conclusions.

2 Code Loading in Web Applications

In this section, we provide background on the mechanics of code loading in Web applications, and illustrate how applications today may take advantage of dynamic code loading to improve their user-perceived performance.

2.1 Background: Mechanics of Code Loading

In its most basic form, the client-side component of a typical distributed Web 2.0 application consists of a number of HTML pages that refer to resources such as images, cascading style sheets (CSS), and JavaScript code. The most natural way to transfer JavaScript files or, indeed, any type of resource is by specifying their names directly in

```
var xhr = new XmlHttpRequest();
// synchronous AJAX call to fetch function foo
xhr.open("http://code.server.com/code=foo", false);
xhr.send(null);

// code is returned from the server as text
var code = xhr.responseText;

// eval is used to get a function closure for foo
var foo = eval(code);
// proceed to use newly loaded function foo
var x = foo(3);
```

Figure 2: Dynamic code loading in an AJAX application

HTML, as shown in Figure 3. When it comes to downloading JavaScript code, this approach causes the Web browser to block until the entire code base is transferred to the client, leading to long pauses in application loading and execution. Before the advent of dynamic HTML, this was the only technique for client-side code loading.

Modern browsers, however, allow for *dynamic code loading*, where resources can be fetched on-demand from a server, using a remote procedure call over HTTP [23]. Just like with data, the code in question may be fetched as a string and then executed using a call to `eval` using the `XmlHttpRequest` object, as shown in Figure 2.

Dynamic code loading enables application architectures in which only a small portion of the code — the basic application framework — is transferred to the client initially. The rest of the code is loaded dynamically at a later point. Some JavaScript programming toolkits have begun to include basic support for dynamic code loading in order to support these scenarios. For example, the Dojo Toolkit [2], modeled after the Java class loader, provides a small bootstrap script `dojo.js` and enables dynamic loading through the library function `dojo.require("...");`.

2.2 Case Studies

Despite this support for dynamic code loading in the underlying JavaScript language and toolkits, building an application that successfully exploits dynamic code loading to improve user-perceived performance is a difficult task. In the next several subsections, we show how today's Web 2.0 applications are structured to load their code and point out opportunities for improvement.

A summary of information about the application below as well as other widely-used Web 2.0 sites is given in Figure 1. It shows the breakdown of (compressed) sizes for different application components (gzip compression is a popular way to reduce resource sizes). As can be seen from the figure, the largest two categories of resources by far are JavaScript code and images. The reader is referred to Section 4 for more detailed information on our benchmarks.

```
<html>
    <head>
        <script src="scripts/schedule.js">
        <script src="scripts/string_library.js">
        <script src="scripts/clouds.js">
        <script src="scripts/bunnies.js">
        <script src="scripts/preload.js">
    </head>
    ...
</html>
```

Figure 3: A static script loading strategy from the Bunny Hunt JavaScript game.

2.2.1 All-at-once Loading: Bunny Hunt

Bunny Hunt, the application with the smallest JavaScript codebase in our suite of benchmarks of only 17 KB, takes the extreme approach to resource loading: in addition to transferring all the application code as part of the application splash screen, images are preloaded as well.

The Bunny Hunt approach to resource usage is fully conservative: the entire network transfer cost is paid upfront. By downloading every single resource, including both JavaScript code and image files while the splash page is loading, page rendering can never block waiting for either of these types of resources. While the opposite extreme would be to download every resource on demand, as it is needed, most applications fall somewhere inbetween.

The Bunny Hunt code base is broken down into five files, each of which is transferred separately, as shown in Figure 3. Notice that this type of declaration precludes JavaScript files from being downloaded in parallel: the browser has to start executing each of JavaScript file in the order they appear on the page. While this may be a reasonable approach when the entire application, containing HTML and images is under 400 KB, for larger benchmarks applications this is probably not the best strategy.

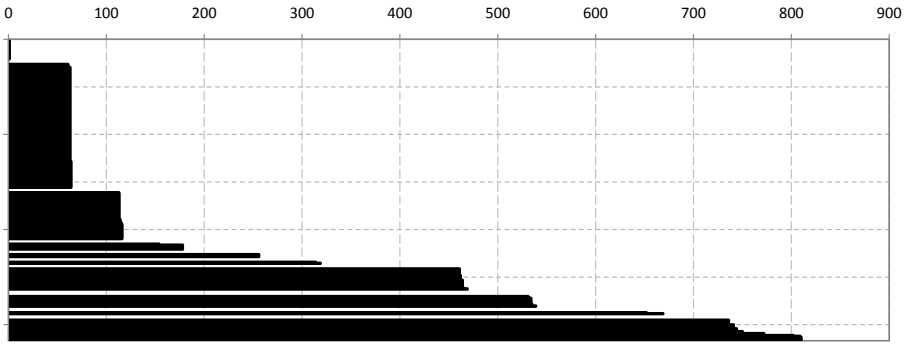

Figure 4: Each line corresponds to an individual function in `mapcontrol.asjx`, a JavaScript file in Live Maps. The length of each line indicates the difference between the available and first-use times. Gaps are inserted to show the natural block structure.

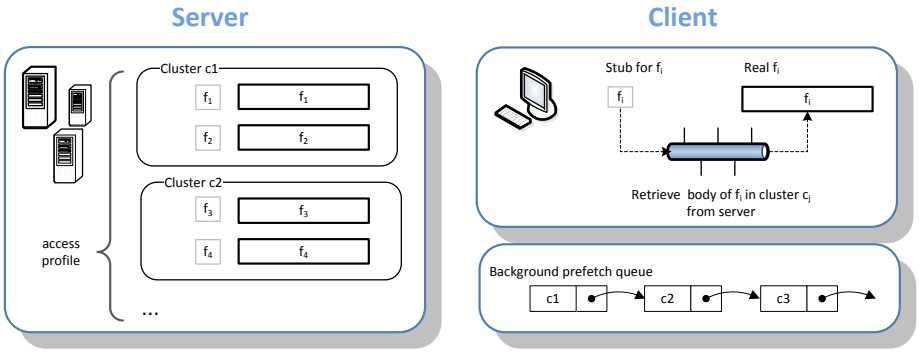

Figure 5: Architectural diagram of DOLOTO.

2.2.2 Dynamic Loading: Pageflakes

A contrast to Bunny Hunt is the Pageflakes application, an industrial-strength mashup page providing portal-like functionality. While the download size for Pageflakes is over 1 MB, its initial execution time appears to be quite fast. Examining network activity reveals that Pageflakes downloads only a small stub of code with the initial page, and loads the rest of its code dynamically in the background.

It is instructive to compare Pageflakes to a functionally similar but architecturally different mashup page called Dropthings. Dropthings is an application created on top of AJAX.NET, a popular AJAX toolkit and is a typical example of a framework-based application. As Figure 1 makes clear, JavaScript code constitutes over 100 KB or about 50% of the entire Dropthings transfer. Even though it is over four times smaller than Pageflakes, its initial execution is noticeably slower. Upon examination, it is apparent that not only does Dropthings download all its code at start-up, but it downloads each of its 12 JavaScript files serially, instead of in parallel.

As illustrated in this comparison of Pageflakes to Dropthings, developers today can use dynamic code loading to improve their web application's performance. However, designing an application architecture that is amenable to dynamic code loading requires careful consideration of JavaScript language issues such as function closures, proper lexical scoping, etc. Moreover, an optimal decomposition of code into dynamically loaded components often requires developers to set aside the semantic groupings of code and instead primarily consider the execution order of functions. Of course, evolving code and changing user workloads make both of these issues a software maintenance nightmare.

The goal of DOLOTO is to automate the process of optimal code decomposition. DOLOTO's processing of application code automatically handles language issues such as closures and scoping; DOLOTO's analysis discovers an appropriate code decomposition for the likely execution order of functions. As a consequence of this sort of automation, developers no longer have to manually maintain the decomposed version of the application as the application or typical usage scenarios change, but simply reapply the analysis and decomposition as necessary.

```
{c_1, ..., c_n, ⊥} = read_clusters();     // read access profile
foreach ( js ∈ application ){              // for each file js
    transfer DOLOTO helper functions
    foreach ( f ∈ in js ){                 // for each function f
        if ( isLarge(f) && (f ∉ c_1) ) {
            replace f with stub for f
        } else {
            transfer f verbatim
        }
    }
}
```

Figure 6: Pseudocode for server-side processing in DOLOTO.

2.2.3 Typical Monolithic Application: Live Maps

Live Maps is an example of a large, feature-rich application, providing multiple-views of maps and satellite photos, driving directions, business search, vector drawing capabilities, advertising, and more. It loads over 200 KB of code compressed on the initial page load. While the entire codebase is downloaded upfront, only a fraction of it is executed on the initial page load. Furthermore, code execution takes place in "bursts", as illustrated in Figure 4: while the entire JavaScript file is available on the client, some functions are executed right away, as indicated by the initial block, some are executed within 100 ms or so. However, many functions are not executed until triggered by user interaction (such as a request for driving directions a map search) and in many common usage scenarios, these functions will not be needed at all. The "ladder pattern" in Figure 4 provides intuition for our function clustering approach used in the training phase of DOLOTO.

DOLOTO primarily targets feature-rich Web applications such as Live Maps. The code base of these applications is growing as they expand to provide functionality once reserved for traditional desktop software. Unfortunately, the latency cost of downloading this additional code is paid whether or not the additional features are used. This suggests that splitting the code of these features out, and dynamically loading them outside the critical-performance-path of initialization is likely to improve initial page loading times.

3 Doloto Architecture

DOLOTO processing consists of two phases, the training and the execution phase described in the rest of this section. The training phase of DOLOTO's processing consists of running the application with its client-side JavaScript component instrumented to collect function-level profile information.The result of this training is an *access profile*, a clustering of original functions by time of their first use. In our implementation, training is performed by observing a user performing a fixed workload, although it is possible to train in a distributed manner, by combining workloads from multiple users with varying workloads, resulting in better code coverage and higher quality access profiles.

In the second phase, DOLOTO proceeds to rewrite existing application JavaScript code based on a given access profile to split the existing code base into small stubs that are transferred eagerly and the rest of the code that is transferred either on-demand on in the background using a prefetch queue. An architectural diagram showing the details of the execution phase of DOLOTO is shown in Figure 5. The basis for our approach is to rewrite every JavaScript function `f` with a stub that in its simplified version looks as follows:

```
function f(){
  var real_f_text = blocking_download("f");    // download function text for "f"
  var real_f_func = this.eval(real_f_text);    // create a function closure
  return real_f_func.apply(this, arguments);   // evaluate closure on arguments
}
```

Helper function `blocking_download` is a blocking function that retrieves the body of function `f` from the server. A network call is made only once per cluster: if the body of `f` has already been transferred to the client either on-demand or background code loading, `blocking_download` returns it immediately. We proceed to `eval` the body at runtime and apply the resulting function to the arguments that are being passed into `f`. We refine and optimize this simplified pseudo-code in Section 3.3.

3.1 Collecting Access Profiles

At its core, our instrumentation approach is based on the ability to parse and instrument JavaScript code and to insert timestamps that allow us to group functions into clusters by the time of their first access. Our instrumentation machinery is based on a proxy-based JavaScript rewriting platform. This approach allows us to use a local proxy to obtain timing information for external Web sites that we do not have access to.

The beginning of every JavaScript function transferred to the browser is instrumented to insert a timestamp as well as the size of the function. Timestamps are subsequently collected by the proxy and post-processed to extract the first-access time ts_i for every function f_i that is observed at runtime. To avoid excessive network traffic that would perturb normal application execution, timestamp data is buffered on the client before being sent over the to training proxy.

The list of timestamps is subsequently sorted and traversed to group functions into clusters $c_1, \ldots, c_n$. As we are traversing the sorted list we are looking to terminate the current cluster c_j at function f_i according to the following criterion:

$$ts_{i+1} - ts_i > T_{gap} \wedge size(c_j) > T_{size},$$

i.e. we should terminate the current list of functions and turn it into a cluster if the time gap between the two subsequent functions exceeds the predefined gap threshold T_{gap} and the size of the current cluster exceeds the predefined size threshold T_{size}. Note that we completely disregard the original decomposition of functions into files: functions from different JavaScript files may and do end up belonging to the same clusters because of temporal proximity to each other.

Note that as with any runtime analysis, a potential weakness of this approach is that some code may not be used for the workload we apply: for instance, if the "help"

functionality of a mapping site is not utilized during the initial page load, functions implementing this functionality will be group into an special cluster $\perp$. As part of the process, map

$$\mathcal{P} : \{f_1, \ldots, f_k\} \rightarrow \{c_1, \ldots, c_n, \perp\}$$

from functions to clusters is saved as the access profile.

3.2 Server-side Code Rewriting

As mentioned above, the basis of our approach is to replace original JavaScript functions with short stubs and then fetch (potentially large) function bodies either on demand or whenever extra bandwidth becomes available. The client-side component of a Web 2.0 application consists of a set of JavaScript files; JavaScript code may also be included directly in HTML, but each inline script block is conceptually treated as a separate file. Each JavaScript file consists of top-level code that is executed unconditionally and a set of function declarations. Each function declaration it its turn may contain top-level code as well as local function declarations.

The pseudocode for our server-side processing is shown in Figure 6. For each file we rewrite with DOLOTO, we start by transferring the necessary helper functions such as `blocking_download`, etc. that are required for dynamic code loading, background code prefetch, and generating stub code on the client. Next, for every function in the file, we decide whether to transfer it verbatim or to replace it with a stub. This decision is based on the length of the function (in practice we only transfer files longer than 50 characters) and whether the function is in the first cluster c_1, which we transfer eagerly, i.e. without resorting to stubbing.

While we implemented DOLOTO as a proxy, in the future we envision server-side deployment or tight integration with the Web server. One option is to perform periodic rewriting of the application code based on the current access profile offline and save the result in order to avoid the latency of the rewriting process being added to the critical path of the application execution.

```
var g = 10;
function f1(){
    var x = g + 1;
    ...
    return ...;
}

function f2(){
    ...
    return ...;
}
```

Figure 7: Example before DOLOTO rewriting.

```
1       var real_f1 = null;
2       function f1(){
3           if(real_f1 == null){
4               guard_cluster_c1();
5               real_f1 = this.eval(func["f1"]);
6               f1 = real_f1;
7           }
8
9           return real_f1.apply(this, arguments);
10      }
11
12      var real_f2 = null;
13      function f2(){
14          if(real_f2 == null){
15              guard_cluster_c1();
16              real_f2 = this.eval(func["f2"]);
17              f2 = real_f2;
18          }
19
20          return real_f2.apply(this, arguments);
21      }
22
23      function guard_cluster_c1(){
24          var xhr = new XmlHttpRequest();
25          xhr.open("http://code.server.com/cluster=c1",
26          /* synchronous AJAX call */         false);
27          xhr.send(null);
28          var code = xhr.responseText;
29          // split code into function bodies
30          foreach(<func_name, func_code> in code) {
31              func[func_name] = func_code;
32          }
33          // empty closure
34          guard_cluster_c1 = function() {};
35      }
```

Figure 8: Rewriting by introducing stubs and a download guard.

3.3 Client-Side Execution

To summarize, the client-side execution of the application is affected by DOLOTO in the following ways:

- When a new JavaScript file is received from the server on the client, we let the browser execute it normally. This involves running the top-level code that is contained in the file and creating stubs for top-level functions contained therein.
- When a function stub is hit at runtime,
  - if there is no locally cached function closure, download the function code using helper function `blocking_download`, apply `eval` to it, and cache the resulting function closure locally;
  - apply the locally cached closure and return the result

- When the application has finished its initialization and a timer is hit, fetch the next cluster from the server and save functions contained in it on the client.

3.3.1 Illustrative Example

We first illustrate DOLOTO rewriting with examples and then describe implementation details and important corner cases of local functions and function closures as well as optimizations to reduce both the runtime overhead as well as the size of the code that needs to be transferred to the client.

Figure 8 illustrates how function rewriting works by showing the result of rewriting global functions `f1` and `f2` shown in Figure 7 that both belong to cluster `c1`. Below we focus on function `f1`; function `f2` is treated similarly:

1. For each function, DOLOTO turns its body into a short stub shown on lines 1–10 (our example uses longer variable names for clarity). The stub first invokes the guard for the cluster the function belongs to (`c1` in this case). This is a blocking action that only returns after the body of the function has been saved in the global associative array `func`.
2. Next, the function body is retrieved as text and evaluated using the `eval` construct of JavaScript. Note that the call to `eval` is performed in the same scope as the original function definition for `f1` on line 5. This way, since the body of `f1` refers to global variable `g`, at the time of applying `eval`, variable `g` will be resolved properly. This is why we cannot, for example, perform the `eval` of the original function body within the guard function and return the closure corresponding to the actual code of `f1`. The result of the `eval` call is saved in global variable `real_f1` so that the `if` body is only entered once per function.
3. Lastly, on line 9 we return the result of applying the real function body stored in `real_f1` to the original set of arguments on object `this`.

3.3.2 Runtime Optimizations

Additionally, for the example above, we can perform the following optimizations at the time of rewriting to reduce the runtime overhead experienced by the rewritten code compared to the original.

Reassigning function value. As an optimization tactic, we assign the closure returned from `eval` to `f1` on line 6. This way, the second invocation of `f1` will go directly to the original code completely circumventing our rewriting. However, things are more complicated in the presence of function aliasing: if a references to `f1` was taken prior to `f1` being executed, then the guarded version of `f1` may still be called through that reference, so it is unsafe to eliminate it completely.

Guard elimination. Note that before existing, `guard_cluster_c1` "eliminates itself" by assigning the empty closure to global variable `guard_cluster_c1` on line 34. This way, the guard body will only be executed *once per cluster*. For instance, if function `f2` is invoked after `f1`, the guard will be a no-op.

```
1           var xhr = new XmlHttpRequest();
2           function next_cluster(){
3               xhr.open("http://code.server.com/next",
4               /* asynchronous AJAX call */     true);
5
6               xhr.onreadstatechange = handle_cluster;
7               xhr.send(null);
8           }
9
10          function handle_cluster(){
11              if (xhr.readyState != 4)  { return; }
12              var code = xhr.responseText;
13              if (code == "") return; // last cluster
14
15              // split code into function bodies
16              foreach(<func_name, func_code> in code) {
17                  func[func_name] = func_code;
18              }
19
20              // go fetch the next cluster
21              next_cluster();
22          }
23
24          // initial invocation of next_cluster
25          // after the document is done loading
26          document.attachEvent("onload", next_cluster);
```

Figure 9: Background code prefetching.

3.3.3 Code Rewriting in JavaScript

In many ways, the example above illustrates the "best case scenario" for our rewriting technique. There are several concerns we have to address when performing function rewriting. Unlike many other mainstream languages, JavaScript allows nested function definitions. Local functions complicate our rewriting strategy, making it necessary to cache real function bodies (`real_f1` and `real_f2` in examples above) in a local context just before the function definition. Also notice that since local declarations may close over variables in the lexical scope, we are careful to perform evaluation of real function bodies in the same context as the original function declaration. Clearly, performing as `eval` in the top-most lexical scope, for example, may create references to undefined variables.

JavaScript allows the developer to define function closures and which can assigned to variables, passed around, and invoked arbitrarily. Unlike regular function definitions, closures are allowed to be anonymous. When rewriting anonymous closures, we have to traverse up the AST to find an appropriate place for introducing cache variables. Furthermore, the optimization of reassigning the function value does not apply to function closures, which often have multiple aliases within the program.

3.3.4 Additional Code Size Optimizations

Note that the stubs shown in the example above tend to still be fairly long. To save extra space, we apply the technique described below that typically reduces the size of a stub from several hundred characters to under 50. A key insight is that JavaScript is a dynamic language that allows function introduction at runtime; we do not have to transfer complete stubs over the network as long as we can generate them on the client. Therefore, we parameterize the text of each stub with its name and argument names and then introduce a helper function `exp`(`function_name`, `argument_names`) to generate the stub body at runtime.

Therefore, in the example in Figure 8, we replace stubs for function `f1` and `f2` as well as the guard for cluster `c1` with

```
eval(exp("f1",""));eval(exp("f2",""));
```

This runtime code generation reduces the download size at the expense of introducing extra runtime overhead for running function `exp` and applying `eval` to the resulting string. Also note that for nested functions, their stubs are introduced at runtime lazily, after the body of containing functions have been expanded. In practice, techniques described in this section save hundreds of kilobytes of JavaScript for large applications such as Live Maps.

3.4 Background Code Prefetch

Background code prefetching allows us to *push* code to the client instead of having the client *pull* code from the server. When translating JavaScript files, we inject prefetch code shown in Figure 9 into each HTML file passed to the client. Our approach relies on the server maintaining per-client status with respect to the code that has already been transferred over. A viable alternative to would be to transfer cluster information to the client so that it would be able to specify to the server which function to fetch. Note that when fetching a cluster it is not necessary to specify the entire cluster: a single function from it will suffice to bring the entire cluster over.

Function `next_cluster` requests the *next* cluster in the access profile that has not yet been transferred over from the server. Function `handle_cluster` is registered as an AJAX callback on line 6 to process the server response to update the global array `func` with the function bodies it retrieved. As the last step, on line 21 `handle_cluster` calls `next_cluster` again. This way, there is a continuous queue of downloads from the server that driven by the client. The initial code request is performed by registering an `onload` handler for the page as shown on line 26.

Cluster $\perp$ which contains functions that are never seen as part of runtime training is never returned by the server eagerly and functions in it may only be downloaded individually on-demand.

4 Experimental Results

In this section, we evaluate the performance of the DOLOTO code splitting approach against the five benchmark applications shown in Figure 10. These applications were

Web application	Application URL	Description
Chi game	`http://chi.lexigame.com`	Online arcade game
Bunny Hunt	`http://www.themaninblue.com/experiment/BunnyHunt`	Online arcade game
Live.com	`http://www.live.com`	Customizable mash-up page
Live Maps	`http://maps.live.com`	Interactive mapping and driving directions application
Google Spreadsheets	`http://spreadsheets.google.com`	Online spreadsheet application

Figure 10: Summary of information about benchmark Web 2.0 applications used in this paper.

Figure 11: DOLOTO training setup.

chosen to represent a range of small to large applications, and to test a range of JavaScript code vs. resource ratios. As suggested in Figure 1, JavaScript code and images are the two largest components for many modern Web 2.0 applications. At one extreme, the relatively small application Bunny Hunt uses very little JavaScript code and many large images. At the other extreme, Google Spreadsheets is composed of very few images and a large JavaScript code base.

4.1 Experimental Setup

The goal of our experimental setup was to evaluate the impact of code splitting on the download size and initial responsiveness of real-world, third-party Web applications, for a variety of realistic network conditions. The setup of our experimental testbed faced several challenges:

Modifying 3rd-party applications: While we propose that DOLOTO be part of server-side Web application deployment strategy, we obviously do not have control over the server-side environments of the applications with which we are experimenting. In order to apply DOLOTO to these applications, we implemented DOLOTO as a rewriting proxy that intercepts the responses from 3rd-party Web servers and dynamically rewrites their JavaScript content using our code splitting policies. In all our experiments, our client-side Web browsers are chained to the proxy implementation of DOLOTO. To accurately simulate a server-side deployment of DOLOTO with off-line rewriting of application code, we need to ensure that our dynamic rewriting is not in the critical path of serving Web pages. Thus, our DOLOTO proxy caches the results of its rewritings such that a second visit to the page is immediately fulfilled without additional processing.

Sites serving multiple versions of Web application code: Web sites frequently serve different versions of their Web applications over time, either as part of a rolling upgrade or as part of a concurrent A/B test of new functionality. To get comparable and consistent results across multiple runs, our experiments rely on training and executing on the same Web application code. To be sure that our experiments are always run against the same version, we deployed the Squid caching proxy [18] that held a single copy of our benchmark Web application code. To ensure that all components of the Web applications were cached, we used an additional HTTP rewriting proxy, Fiddler [8], that forces all components of a Web application to be cache-able by modifying the HTTP cache-control headers set by the original Web site.

		Code coverage in training				Function characterization				Cluster statistics		
						Size distribution						
Web application	**Download size, in KB**	**Number functions**	**%**	**Total size, in KB**	**%**	**<100**	**100-200**	**200-500**	**>500**	**Number of clusters**	**Functions per cluster**	**Average size**
Chi game	104	103	29%	43	41%	22	26	28	27	7	3/14/43	6.2
Bunny Hunt	16	22	44%	10	60%	5	0	9	8	3	2/7/19	3.3
Live.com	1,436	689	21%	572	39%	203	149	165	172	14	5/49/461	40.9
Live Maps	1,909	803	16%	835	43%	284	188	177	154	12	6/66/689	69.7
Google Spreadsheets	499	794	24%	179	35%	442	156	121	75	15	3/52/648	12.0

Figure 12: Training statistics for our benchmark applications.

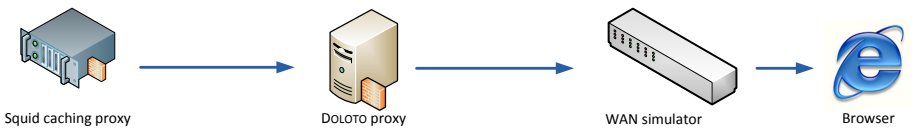

Figure 13: DOLOTO testing setup.

Simulating realistic network conditions: In order to collect evaluate download times in a realistic range of network conditions, we used a wide-area network simulator that provides control over the effective bandwidth, latency, and packet loss rates of a machine's network connection. We use this network simulator to simulate 3 different environments: a Cable/DSL connection with a low-latency network path to a Web site (300 kbps downstream bandwidth and 50 ms round-trip latency); a Cable/DSL connection with a high-latency network path (300 kbps downstream bandwidth and 300 ms round-trip latency); and a 56k dial-up connection (50 kbps downstream bandwidth and 300 ms round-trip latency).

Our resulting setup for training, using Fiddler, Squid, and DOLOTO is shown in Figure 11. The DOLOTO proxy was running on a machine with a dual Intel Xeon, 3.4GHz CPU, with 2.5GB of RAM. Our experimental setup for evaluating network download times adds a wide-area network simulator, and is shown in Figure 13. The client-side browser used is Firefox 2.0 and was running on a Pentium 4 3.6 GHz machine equipped with 3 GB of memory running Windows Vista. The physical network connection between all our test machines is a 100 Mb local area network over a single hub.

4.2 Training Phase Statistics

To train the clusters and create the access profiles for a Web application, we collected a profile of several minutes of each Web application's execution under a manual workload that exercised a variety of each application's functionality. For example, the manual workload for Bunny Hunt and the Chi game consists of playing the game and the workload for `http://maps.live.com` consists of browsing and searching through the map.

A summary of results for the training phase is shown in Figure 12. Column 2 shows the total (uncompressed) download size for each application in our benchmark suite. Columns 3–6 show information about the code coverage observed during our training run, detailing the number of functions called during the run (absolute number and percentage in columns 3–4) and the *size* of these functions (absolute number and percentage in columns 5–6). Columns 6–9 show a distribution of function sizes that we have observed at runtime. While small functions are quite common, especially in sites such as GMail and Google Maps that introduce them for obfuscation purposes, there are quite a number of large functions as well, indicating the potential to benefit from removing functions from the initial download.

Finally, columns 10–12 show information about the clusters we constructed. Column 10 shows the minimum-average-maximum number of functions per cluster. As

			Application size, in bytes								
Web application	**Functions rewritten**		**Regular**			**Crunched**			**Crunched and Gzipped**		
			Original	DOLOTO	**Savings**	**Original**	DOLOTO	**Savings**	**Original**	DOLOTO	**Savings**
Chi game	202	71%	124,647	69,073	45%	125,045	69,469	44%	34,227	21,004	39%
Bunny Hunt	24	61%	17,216	7,692	55%	17,371	7,841	55%	4,836	3,033	37%
Live.com	1,680	58%	882,409	472,680	46%	882,438	472,706	46%	220,544	129,278	41%
Live Maps	1,463	42%	1,125,600	617,171	45%	1,125,618	617,183	45%	270,644	155,992	42%
Google Spreadsheets	1,382	48%	654,142	402,010	38%	654,192	402,060	38%	180,367	96,452	46%

Figure 14: Size reduction after DOLOTO rewriting for our benchmarks.

Web application	50 kbs/300 ms			300 kbs/300 ms			300 kbs/50 ms		
	Orig.	DOLOTO	**%**	**Orig.**	DOLOTO	**%**	**Orig.**	DOLOTO	**%**
Chi game	37	37	0%	13	15	13%	8	8	0
Bunny Hunt	100	92	8%	43	41	5%	22	22	0%
Live.com	99	82	17%	31	28	10%	18	13	28%
Live Maps	155	112	28%	31	23	26%	26	19	27%
Google Sp'sheet	58	45	22%	20	20	0%	18	11	39%

Figure 15: Reduction in execution times achieved with DOLOTO. Orig. is the original download time, DOLOTO is the time to download the whole app in the background, and % is the percentage difference.

can be seen from the table, it is fairly typical to have several dozen clusters for the larger applications, with some clusters being quite sizable, containing several hundred functions tends of kilobytes of code in the case of Google Maps and Live Maps, respectively. Note that the number of clusters is quite sensitive to the threshold selection. For these results, we used a threshold of 25 ms for the gap between first-run times for functions and a minimum cluster size threshold of 1.5 KB. We created at least one cluster for application frame for applications that contained multiple `FRAME` or `IFRAME` tags, which explains a cluster of size 1 KB in Pageflakes. Furthermore, for the purposes of measuring download size and time improvements, we ensured that all the functions used during page initialization were included in the initial cluster together.

4.3 Execution Phase Statistics

Figure 14 shows the reduction of size achieved with DOLOTO rewriting for our application benchmarks. Columns 2 and 3 show information about the number of percentage of the functions that are rewritten to insert stubs. Since the first cluster is not rewritten and pushed to the application verbatim, less than 100% of all functions end up being stubbed. Columns 4–6 show the size of the regular (uncompressed) code in its original version, the size of the initial DOLOTO download that includes all the stubs that

Web application	50 kbs/300 ms			300 kbs/300 ms			300 kbs/50 ms		
	Orig.	DOLOTO	**%**	**Orig.**	DOLOTO	**%**	**Orig.**	DOLOTO	**%**
Chi game	37	40	8%	15	15	0%	8	17	113%
Bunny Hunt	100	100	0%	42	43	2%	22	22	0%
Live.com	99	216	118%	31	44	42%	18	36	100%
Live Maps	155	210	35%	31	57	84%	26	52	100%
Google Sp'sheet	58	70	21%	20	24	20%	18	18	0%

Figure 16: Background code loading overhead. Orig. is the original download time, DOLOTO is the time to download the whole app in the background, and % is the percentage difference.

are sent to the client initially, and the resulting space savings. Columns 7–10 show the same numbers with the code having been run through a JavaScript crunching utility that removes superfluous whitespace — a common strategy for optimizing released JavaScript code. The tool configuration we used did not perform any additional optimizations such as shortening local variable identifiers. Finally, columns 11-14 show the same set of numbers after the code has been crunched *and* run through a gzip compression utility. Gzip compression is a common and perhaps the easiest strategy for reducing the amount of data transferred over the network and, as such, is used quite widely by the sites we chose as our benchmarks.

In addition to size reduction measurements, we also performed detailed experiments with several representative benchmarks to determine the effect of code size reduction on the overall application execution time for a range of network parameters, as shown in Figure 15. For each group of columns in Figure 15, we show the original execution time, the time with DOLOTO, and the percentage of time savings. Clearly, whether the size reduction is accomplished by DOLOTO will translated into execution time reduction depends heavily on application decomposition (more information about sizes of individual components of our benchmarks is shown in Figure 1). It is common to have images and JavaScript code as the biggest application components; below we consider applications with different ratios of the two.

It is not too surprising that, as an application whose download is dominated by images, Bunny Hunt does not show any significant improvements with DOLOTO. On the other hand, mash-up site Live.com, which has JavaScript as its most significant download component, shows pretty significant speed-ups, especially in the case of a low-latency high-bandwidth connection. For high-latency connections, the time savings are tangible, but not as significant because the execution time is dominated by the need to connect to many servers to fetch data to be shown on the mash-up page.

Live Maps shows 26-28% improvements for a range of network conditions, with dozens of seconds being saved on the slowest connection. This is quite impressive given that a significant portion of the application execution is spent on retrieving map images. However, as Figure 14 shows, these time savings can be explained by the fact that about 45% of the application code is not being transferred in the DOLOTO version. Time savings are most significant for Google Spreadsheet, in which code is the most significant download component. Because the entire application is under 200 KB in size and the image component is quite small, savings accomplished with DOLOTO result in noticable speedups. However, on a 300 ms latency connection, third-party server requests that are used for analytics collection dominate the download time, masking the savings achieved with DOLOTO.

Figure 16 shows the additional time it takes to download the entire codebase of an application with background downloading. In general, we see that the additional time to download an application is roughly proportional to the benefit received from code splitting. The intuition behind this is that the total download size, and hence the download time, is increased by the number of stubs and code added to remove functions from the critical download path. Because users can interact with and use an application while background downloading is occurring, we believe this trade-off of longer total download time for shorter latency until a page responds to user interactions is more than worthwhile.

5 Related Work

While much work has been done on improving server-side Web application performance and reducing the processing latency [13, 19, 20], recent studies of modern Web 2.0 applications indicate that front-end execution contributes 95% of execution time with an empty browser cache and 88% with a full browser cache [17]. Moreover, browser-side caching of Web content is less effective than previously believed because about 40% of users come with an empty cache [16]. This, along with a trend towards network delivery of increasingly sophisticated distributed Web applications, as exemplified by technologies such as Silvelight, highlights the importance of client-side optimizations for archieving good end-to-end application performance.

Other than the MapJAX project's work on data prefetching in Web 2.0 applications [11], we are not aware of research directly pertaining to responsiveness of Web 2.0 applications, though several projects have focused on software that is delivered over the network. In particular, Krintz et al. propose a technique for splitting and prefetching Java classes to reduce the application transfer delay [7]. Class splitting is a code transformation that involves breaking a given class into part: hot and cold, depending on usage patterns observed a profile time. The cold part is shipped to the client later in a demand-driven fashion. Our profile construction approach may be seen as an extension of their technique, in particular, the code clusters we identify represent "degrees of urgency": the first cluster must be transferred right away, while others can be transferred later so their transfer is overlapped with client-side execution. Finally, code whose execution was *not* observed in our profile runs often constitutes a significant portion of the application as explained in Section 2.

Other researchers have focused on reducing the amount of code that is shipped over the wire, most notably in the case of extracting Java applications [21, 22]. There are several distinguishing characteristics between that work and ours. First, with some notable exceptions, JavaScript applications have not yet taken advantage of library-based application decomposition. Exceptions include reliance on Ajax libraries such as AJAX.NET, the Dojo Toolkit, and others, which suggests that going forward, the issue of application extraction may become important once again. Second, the reason for performing extraction was the need to minimize space requirements for applications that are designed to be deployed in embedded settings such as J2ME.

Networking tools such as the RabbIT proxy [12] focus on reducing the total download size by performing image and HTML compression on the wire. It is worth pointing out that many of these techniques are already integrated into commercial-grade sites. However, even with these measures in place, performed either by the server or by a proxy on the network, there is still room for restructuring the code in a manner proposed by DOLOTO.

Recently developed Web 2.0 frameworks such as the Dojo toolkit [2] support explicit code loading of JavaScript in a manner similar to languages with dynamic code loading such as Java and C# [3, 5, 9]. This approach relies on the developer breaking the application into meaningful pieces, as opposed to our work that focuses on existing large applications and can work with them without any modifications. In fact, Web 2.0 application performance guides suggest decomposing the application into meaningful pieces manually [17]. Our work can be seen as an attempt to automatically introduce

dynamic code loading for legacy applications.

The BrowserShield, CoreScript and AjaxScope projects use automatic JavaScript rewriting to enforce security policies and monitor the runtime behavior of JavaScript applications [6, 15, 24]. In contrast, DOLOTO uses JavaScript code rewriting for program optimization purposes.

6 Future Work

In this paper we have described a basic code splitting approach. While it results in significant download size and time savings in practice, there are ways in which it can be further enhanced. We list some of the possible future research directions below.

Static analysis to remove guards: currently, our approach described in Section 3.2 conservatively assumes that every single function needs to be guarded, requiring a lot of guards to be introduced. With some static analysis, however, these guards can be optimized away. Indeed, if we construct a call graph of the application and build dominators for it [1], clusters may be arranged so that a guard for every f_1 dominating f_2, `guard_f1` fetches the body of f_2. The fact that f_1 dominates f_2 implies that there is no way to invoke f_2 without calling f_1 first, making our approach sound. With this technique, stubs for many functions would not be transferred at all, unless they become required by a higher-level guard; currently, we conservatively assume that any function in the program may be called.

Static analysis instead of runtime training: In fact, given an application call graph, we may even be able to do away with the training stage of our approach entirely: we can use trees in the dominator forest as clusters that we pass to the second stage of DOLOTO. Of course, the main obstacle to call graph construction or, indeed, most forms of static analysis in JavaScript is the presence of `eval` statements. There are applications, especially those that are generated from tools such as the GWT [4] or Volta [10] that either do not use `eval` or use in is a controlled manner that is amendable to static analysis.

Continuous feedback loop. Currently, the DOLOTO approach is static, with the training and execution phases being entirely separate. It is possible, however, have a feedback loop, in which additional instrumentation is added to a deployed application to update access profiles in real time. To reduce the effect of the instrumentation overhead, we can only send instrumented application version to a fraction of users or we can instrument a single JavaScript file per user, etc.

Context-sensitive access profiles: the current training approach creates a global set of clusters. However, for large applications, it might make sense to have a finer grained techniques, where, depending on where the user is in the application, different portion of the application would be loaded. For instance, there would be different access profiles for the 2D and 3D functionality of an online mapping application. Depending on execution context, different code would be fetched with both on-demand loading and background prefetching.

7 Conclusions

This paper proposes DOLOTO, a system for splitting client-side code of large modern Web 2.0 applications that often contain hundreds of kilobytes of JavaScript code. DOLOTO consists of a training phase that groups code according to the temporal access patterns and a rewriting phase, where the original code is rewritten to contain stubs that fetch the actual code on demand.

When integrated with the server, DOLOTO results in a significant reduction of the amount of code that is necessary for the application to execute, leading to smaller code downloads and more responsive applications.

Our experiments show that DOLOTO reduces the size of the JavaScript code component by as much as 50% and execution times by as much as 39%, with time savings over 20% being common. We expect code splitting to be a key enabler Web 2.0 applications to continue growing in size and sophistication without placing undue code download burden on the application developer.

References

[1] Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[2] Dojo Foundation. Dojo, the JavaScript toolkit. `http://dojotoolkit.org`, 2007.

[3] M. Franz. Dynamic linking of software components. *Computer*, 30(3):74–81, Mar 1997.

[4] Google Web toolkit. `http://code.google.com/webtoolkit`.

[5] T. Jensen, D. Le Métayer, and T. Thorn. Security and dynamic class loading in Java: A formalization. In *Proceedings of the International Conference on Computer Languages*, page 4, 1998.

[6] Emre Kıcıman and Benjamin Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of Web 2.0 applications. In *Proceedings of Symposium on Operating Systems Principles*, October 2007.

[7] Chandra Krintz, Brad Calder, and Urs Hölzle. Reducing transfer delay using Java class file splitting and prefetching. In *OOPSLA*, pages 276–291, 1999.

[8] Eric Lawrence. Fiddler: Web debugging proxy. `http://www.fiddlertool.com/fiddler/`, 2007.

[9] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 36–44, 1998.

[10] Erik Meijer. Democratizing the cloud. In *OOPSLA Companion*, pages 858–859, 2007.

[11] Daniel S. Myers, Jennifer N. Carlisle, James A. Cowling, and Barbara H. Liskov. MapJAX: Data structure abstractions for asynchronous Web applications. In *Proceedings of the Usenix Technical Conference*, June 2007.

[12] Robert Olofsson. RabbIT proxy for a faster Web. `http://rabbit-proxy.sourceforge.net/index.shtml`, 2006.

[13] Giovanni Pacifici, Mike Spreitzer, Asser Tantawi, and Alaa Youssef. Performance management for cluster based Web services. Technical report, IBM Research, 2003.

[14] Pew Internet and American Project. Home broadband adoption 2007. `http://www.pewinternet.org/pdfs/PIP_Broadband\%202007.pdf`, 2007.

[15] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proc. OSDI*, 2006.

[16] Steve Souders. High performance Web sites. `www.stevesouders.com/docs/fowa.ppt`, 2007.

[17] Steve Souders. *High Performance Web Sites: Essential Knowledge for Front-End Engineers*. O'Reilly Media, Inc., 2007.

[18] Squid developers. Squid Web proxy cache. `http://www.squid-cache.org`, 2006.

[19] Christopher Stewart and Kai Shen. Performance modeling and system management for multi-component online services. In *USENIX Symposium on Networked Systems Design and Implementation*, 2005.

[20] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. A scalable application placement controller enterprise data centers. In *WWW 2007: Track: Performance and Scalability*, May 2007.

[21] Frank Tip, Peter F. Sweeney, and Chris Laffra. Extracting library-based Java applications. *Commun. ACM*, 46(8):35–40, 2003.

[22] Frank Tip, Peter F. Sweeney, Chris Laffra, Aldo Eisma, and David Streeter. Practical extraction techniques for Java. *ACM Transactions of Programming Languages and Systems*, 24(6):625–666, 2002.

[23] Wikipedia. XMLHttpRequest. `http://en.wikipedia.org/wiki/XMLHttpRequest`.

[24] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *Proceedings of the Conference on the Principle of Programming Languages*, January 2007.