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 to as AJAX to push application execution to the client web browser. This 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[1], an optimization tool for Web 2.0 applications. DOLOTO analyzes application workloads and *automatically* rewrites the existing application code to introduce dynamic code loading. 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 implementations are transfered lazily in the background or, at the latest, on-demand on first execution of a particular application feature. Moreover, code that is rarely executed is rarely downloaded to the user browser. Because DOLOTO significantly speeds up the application startup and since subsequent code download is interleaved with application execution, applications rewritten with DOLOTO appear much more responsive to the end-user.

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 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. DOLOTO especially shines on wireless and mobile connections, which are becoming increasingly important in today's computing environments.

[1] DOLOTO stands for DOwnLOad Time Optimizer.

SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA

Categories and Subject Descriptors

D.3.4 [**Processors**]: Optimization

General Terms

Web Programming, Optimizations, Code loading

Keywords

dynamic code loading, AJAX, responsiveness

1. INTRODUCTION

Over the last several years, we have witnessed 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 applications, a key challenge thus lies in maintaining and improving application responsiveness despite increased code size.

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 [15].) 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 pay per byte instead of a flat rate.

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 either on-demand

or in the background. 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. 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 Doloto

In this paper, we propose DOLOTO, 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. DOLOTO takes as input the existing client-side code of a Web application and traces of application workloads, and then proceeds to output a rewritten version of application code.

Code splitting in DOLOTO is performed at the level of individual JavaScript functions, which are clustered together to form an *access profile* computed based on function access times computed in a *training phase* with the help of runtime instrumentation. Training is performed entirely on the client-side without the need to access the server. This allows for a quick estimation of potential DOLOTO improvements without affecting the existing deployment.

In the *execution phase*, access profiles guide the process of on-demand code loading. Additionally, code is transferred to the client via a background prefetching queue. 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 the `eval` construct. In particular, to preserve the lexical scoping of function definitions in the original program, we still must 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.

We envision DOLOTO as a deployment-time tool that "repackages" existing Web application code to improve the end-user responsiveness. The developer may experiment several different access profiles and even deploy them in parallel to measure their effectiveness. Given an access profile approved by the developer, DOLOTO deploys a statically rewritten, optimized version of the original application on the server. We envision re-training and rewriting to only be necessary whenever there is a new application release of if the typical workloads change drastically.

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 are particularly pronounced for slower, but increasingly common and important types of connections, such as wireless and mobile, where the execution penalty is especially high if the entire code base is blindly transferred to the client, especially if the user is paying for their connectivity per byte transferred. Moreover, in the international setting, achieving better application responsiveness often means choosing between building more data centers and creating more performant applications, and a tool like DOLOTO helps the developer with the latter.

In our experiments, DOLOTO reduces the size of 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.

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. Effectively, DOLOTO introduces dynamic code loading into applications that have been designed without it in mind.
- We propose a runtime training technique and a clustering scheme for the collected data that automatically groups functions into clusters roughly corresponding to individual high-level application features based on function access times.
- We describe a code rewriting strategy that breaks the code of JavaScript functions into small stubs that are transferred to the client eagerly and fetch remaining function bodies at runtime on-demand, without requiring application-specific knowledge or changes to existing code.
- 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 runtime 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. OVERVIEW

In this section, we provide background on the mechanics of code loading in Web applications, and illustrate how applications today take advantage (or not) of dynamic code loading to improve their user-perceived performance.

2.1 Mechanics of Dynamic 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 resource is by specifying their names directly in HTML. 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 [25]. Just like with data, the code in question may be fetched as a string and then executed using `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

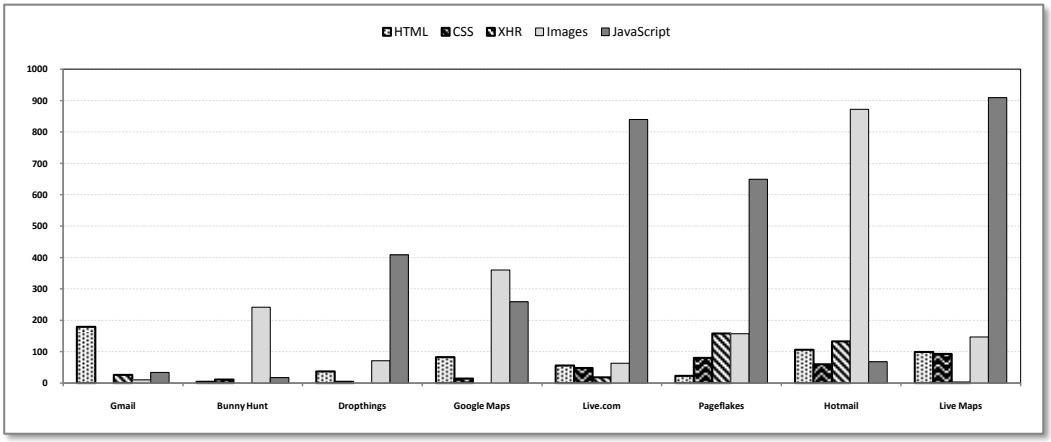


Figure 1: Size breakdown of different Web 2.0 application components. Provided sizes are given in KB after gzip compression.

```
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 downloading of JavaScript 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 a library function `dojo.require("dojo.widget.*")`.

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.

2.2 Dynamic Code Loading in Practice

It is important to point out that JavaScript is often the dominating component of a Web 2.0 application download. To illustrate this point, we have collected size statistics for a range of popular applications, as shown in Figure 1. The figure shows the breakdown of (compressed) sizes for different application components. The largest two categories of resources by far are JavaScript code and images. In the remainder of this section we describe how some representative applications address the issue of code loading.

2.2.1 All-at-once Loading: Bunny Hunt

Bunny Hunt is an application with a relatively small JavaScript code base of 17 KB. This application takes an extreme approach to resource loading. During the application splash screen, it preloads all the relevant code and images. Essentially, Bunny Hunt's 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, most applications fall somewhere in between.

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. As illustrated by Pageflakes, 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, 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.

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 or over 900 KB uncompressed on the initial page load. While the entire code base is downloaded upfront, only a fraction of it is executed on the initial page load. Furthermore, code execution takes place in "bursts": 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. Many functions are not executed until triggered by user interaction (such as a request for driving directions, clicking a button, or performing map search) and in many common usage scenarios, these functions will not be needed at all.

3. DOLOTO ARCHITECTURE

The goal of DOLOTO is to automate the process of optimal code

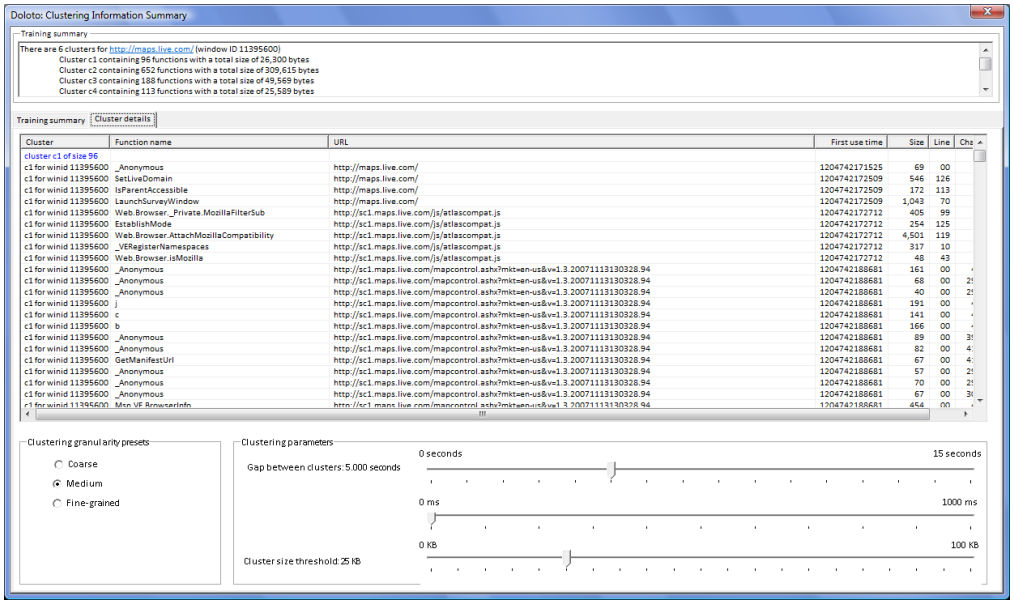


Figure 3: DOLOTO training tool that allows the developer to adjust clustering settings and training thresholds.

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 re-apply the analysis and decomposition as necessary.

DOLOTO's 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.

3.1 Overview

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.

Access profiles may be either static, determined during initial training, or dynamic, being updated continuously and adjusting to operating conditions during application deployment. In DOLOTO, we present the profile information to the developer so that they can tweak the training parameters and review the resulting clusters. Notice that depending on how the application is executed, different cluster decompositions may make sense: for an application that is deployed on mobile devices, small clusters are appropriate, whereas going to the server to fetch a cluster consisting of only several kilobytes of code is probably wasteful for a Web application running on a desktop computer. Next, DOLOTO proceeds to statically 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 or in the background using a prefetch queue.

The basis for our approach is to rewrite every JavaScript function `f(x)` with a stub that in its simplified version looks as follows:

```
function f(x){
    var real_f_text = blocking_download("f");
    var real_f_func = eval(real_f_text);
    return real_f_func.apply(this, arguments);
}
```

Where `blocking_download` is a synchronous call 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 3.3.1. The execution phase of DOLOTO is illustrated in Figure 5.

3.2 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 provided by AjaxScope [7]. This approach allows us to use a local proxy to obtain timing information for external sites that we do not have access to. The beginning of every function is instrumented to record the timestamp as well as the size of the function. At runtime, timestamps are collected by the instrumentation DOLOTO 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, timestamp data is buffered on the client before being sent over to the training

proxy. We refer the interested reader to the AjaxScope paper for details of JavaScript instrumentation.

The list of timestamps is 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. 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 disregard the original decomposition of functions into files: functions from different JavaScript files may and do end up belonging to the same cluster 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 an online mapping application is not utilized during the training run, 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.

In practice, the cluster decomposition changes drastically depending on the threshold values. In our experiments in Section 4 we favored threshold settings that produced about a dozen clusters that roughly correspond to high-level application activities. For instance, the activities of the initial page load, double-clicking on the map, moving the map around, asking for directions, printing the map for Live Maps may each be grouped in their own cluster.

However, for an application that is likely to run in a mobile setting where the user might be paying for byte transferred, producing many clusters of a few kilobytes each is actually a good idea. It is also common to have slightly different versions of the same application for different browsers, so performing training for each browser separately, and then using the appropriate application version based on the execution environment is the right approach.

Examples above illustrate that there is no "perfect" access profile and that the access profile should be customized based on how the application is likely to be used. Moreover, a small benchmark can be injected into the beginning of the application run to determine the network and CPU conditions for a particular user; this information can be cached on the client for subsequent application runs. Based on this data, the application will proceed to download one of several application versions. However, unlike the current practice of developing a separate "mobile" version of an application, all versions would be based on the same code.

3.3 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 in its turn may contain top-level code as well as local function declarations.

The pseudo-code for our server-side processing is shown in Figure 4. For each file we rewrite with DOLOTO, we start by injecting 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 rewrite functions that are longer than 50 characters) and whether the function is in the first cluster c_1, which we transfer eagerly, i.e. without stubbing.

3.3.1 Client-Side Execution

The client-side execution of the rewritten application is affected by DOLOTO in the following way.

- 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 expanding stubs that correspond to declaration of top-level functions contained therein, as further explained in Section 3.3.4.
- 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.

3.3.2 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 7 illustrates how function rewriting works by showing the result of rewriting global functions `f1` and `f2` shown in Figure 6 that both belong to cluster `c1`. In our discussion 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`.

```
// read access profile
{c_1, ..., c_n, ⊥} = read_clusters();

// for each file js
foreach ( js ∈ application ) {
    transfer DOLOTO helper functions

    // for each function f
    foreach ( f ∈ in js ){
        if ( isLarge(f) && (f ∉ c_1) ) {
            replace f with stub for f
        } else {
            transfer f verbatim
        }
    }
}
```

Figure 4: Pseudo-code for server-side processing in DOLOTO.

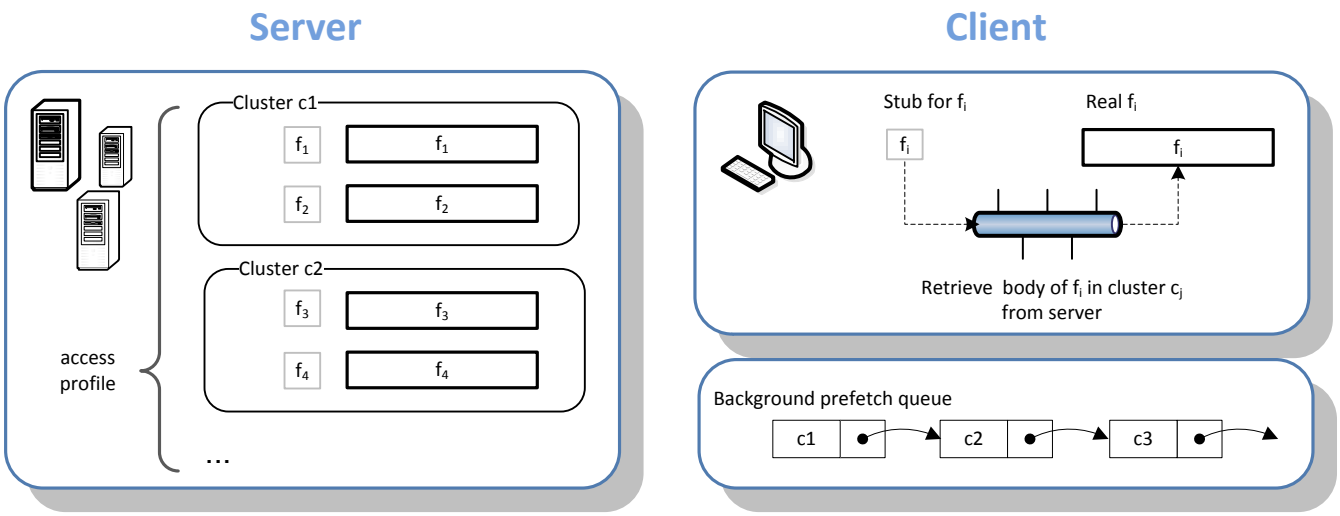


Figure 5: Execution phase of of DOLOTO: the left hand side of the picture shows code located on the server migrating to the client, shown on the right hand side. The execution phase relies on profiles computed during the training phase.

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

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

Figure 6: Example before DOLOTO rewriting.

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 (variable `y` in this case) on object `this`.

3.3.3 *Runtime Optimizations*

Additionally, for the example above, we 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 obtained prior to `f1` being executed,

```
1   var g = 10;
2   var real_f1 = null;
3   function f1(y){
4       if(real_f1 == null){
5           guard_cluster_c1();
6           real_f1 = eval(func["f1"]);
7           f1 = real_f1;
8       }
9
10      return real_f1.apply(this, arguments);
11  }
12
13  var real_f2 = null;
14  function f2(z){
15      if(real_f2 == null){
16          guard_cluster_c1();
17          real_f2 = eval(func["f2"]);
18          f2 = real_f2;
19      }
20
21      return real_f2.apply(this, arguments);
22  }
23
24  function guard_cluster_c1(){
25      var xhr = new XmlHttpRequest();
26      xhr.open("http://code.server.com/cluster=c1",
27        /* synchronous AJAX call */         false);
28      xhr.send(null);
29      var code = xhr.responseText;
30      // split code into function bodies
31      foreach(<func_name, func_code> in code) {
32          func[func_name] = func_code;
33      }
34      // empty closure
35      guard_cluster_c1 = function() {};
36  }
```

Figure 7: Rewriting by introducing stubs and a download guard.

then the guarded version of `f1` may still be called through that reference, so it is unsafe to eliminate it completely.

Guard elimination. 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

```
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 8: Background code prefetching.

body will only be executed *once per cluster*. For instance, if function `f2` is invoked after `f1`, the guard will be a no-op.

Another potential optimization opportunity not explored in the example above involves eliminating blocking calls to the server by converting the program into continuation-passing style. While it is easy to perform an asynchronous server call and to register a callback, the difficulty lies in the fact that we still need to evaluate the code in the proper lexical scope when it arrives, which is not really possible within the callback. A notable exception is global (or top-level) functions, where blocking calls may indeed be eliminated.

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.

In the example in Figure 7, we would 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.

3.3.5 Code Rewriting Caveats

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 an `eval` in the top-most lexical scope, for example, may create references to undefined variables.

JavaScript allows the developer to define function closures, which may be 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.

It is important to recognize that our rewriting relies of the abstraction of functions that are entities that may only be created, assigned, or applied. However, if the original program, for instance, chooses to examine the code of a function by calling `f.toString`(), clearly, the stubbing we perform is going to produce unexpected results. Fortunately, we have not encountered cases such as this is practice.

3.4 Background Code Prefetching

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 8 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 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.

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 function `handle_cluster` calls `next_cluster` again. This way, there is a continuous queue of downloads from the server that is 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 eagerly returned by the server.

Note that we are careful to only download one cluster at a time. This is because most browsers only allow a total of two open connections per server at a time. We do not want to use up all the connectivity just by downloading code. Newer browsers, however, typically increase this threshold to allow for more connections. Viable alternatives to "constant" code prefetching include putting fetching the next cluster based on a timer event or perhaps even detecting periods of user inactivity by intercepting UI events.

4. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of DOLOTO's code splitting against the five benchmark applications shown in Figure 9. These applications were chosen to represent a range of small to large applications, and to test a range of code vs. resource demands. At one extreme, the relatively small application Bunny Hunt uses

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

very little JavaScript code and many images. At the other extreme, Google Spreadsheets is composed of very few images and a large body of JavaScript code.

4.1 Experimental Setup

The goal of our experiments 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. When setting up an experimental testbed, we faced several challenges:

Modifying third-party applications: While it is our proposal that DOLOTO become part of server-side Web application deployment, we do not have any 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 third-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 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.

Sites serving multiple versions of Web application code: Web applications frequently serve different versions of their code 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, 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 [19] 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 [9], 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.

Simulating realistic network conditions: In order to collect execution times for 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 three 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 training setup uses Fiddler, Squid, and DOLOTO as shown in Figure 10(a). The DOLOTO proxy is running on a machine with a dual Intel Xeon, 3.4GHz CPU, with 2.5GB of RAM. The client-side machine is a Pentium 4 3.6 GHz machine equipped with 3 GB of memory running Windows Vista with Firefox 2.0 used as the browser. The physical network connection between all our test machines is a 100 Mb local area network over a single hub.

The corresponding testing setup is shown in Figure 10(b). We first populate a Squid cache running a workload through DOLOTO so that the DOLOTO-processed version of the application is saved in the cache. We then put a a wide-area network simulator between the Squid cache and the browser to evaluate a range of network conditions and replay the same application workload.

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 `maps.live.com` consists of browsing and searching through the map.

A summary of results for the training phase is shown in Figure 11. 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 7–10 show a distribution of function sizes that we have observed at runtime. While small functions are quite common, especially in obfuscated sites that deliberately introduce them, there are quite a number of large functions as well, indicating the potential to benefit from removing functions from the initial download.

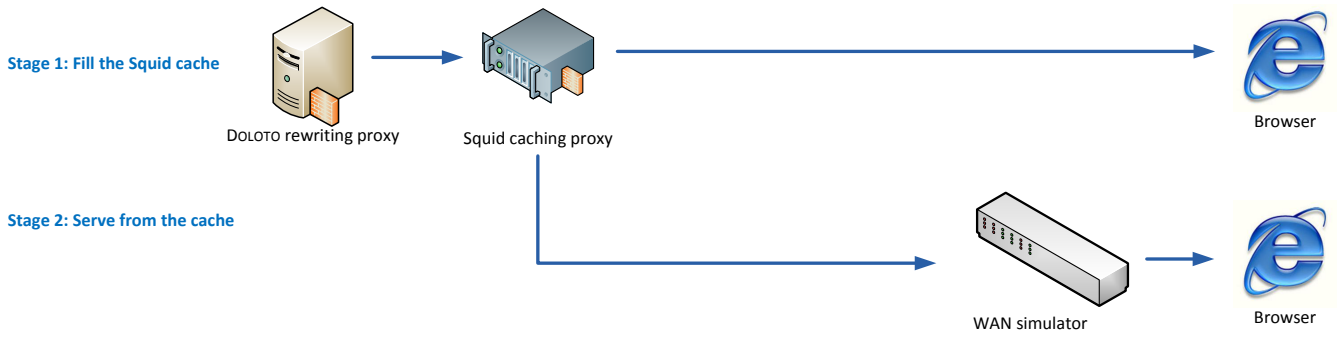
Finally, columns 11–13 show information about the clusters we constructed. Column 11 shows the minimum-average-maximum number of functions per cluster. As can be seen from the table, it is fairly typical to have a dozen clusters for the larger applications, with some clusters being quite sizable, containing several hundred functions and tens of kilobytes of code in the case of larger applications. Also, it is quite common for a cluster to contain functions from more than one file. This demonstrates our reliance on realistic workloads to re-compose the code instead of the initial code decomposition provided by the developer. Column 13 shows the average cluster size, in KB.

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. 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 12 shows the reduction of size achieved with DOLOTO rewriting for our application benchmarks. Columns 2 and 3 show

(a) training setup



(a) testing setup

Figure 10: (a) DOLOTO training setup and (b) testing setup described in Section 4.1.

information about the number of percentage of the functions that are rewritten to insert stubs. Since the first cluster is not rewritten and is 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 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 it is used 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 13. For each group of columns in Figure 13, 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 12 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 14 shows the additional time it takes to download the entire code base 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 [14,

Web application	Download size, in KB	Code coverage in training: Number functions	%	Total size, in KB	%	Function characterization: Size distribution (characters) <100	100-200	200-500	>500	Cluster statistics: Number of clusters	Functions per cluster	Average size, in KB
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 11: Training statistics for our benchmark applications.

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

Figure 12: Size reduction after DOLOTO rewriting for our benchmarks.

Web application	50kbs/300ms Orig.	Dol.	%	300kbs/300ms Orig.	Dol.	%	300kbs/50ms Orig.	Dol.	%
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 13: Reduction in execution times achieved with DOLOTO. Orig. is the original download time, Dol. is the time to download the whole application in the background, both measured in seconds, and % is the percentage difference.

20, 21], 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 [18]. Moreover, browser-side caching of Web content is less effective than previously believed because about 40% of users come with an empty cache [17]. This, along with a trend towards network delivery of increasingly sophisticated distributed Web applications, as exemplified by technologies such as Silverlight [12], 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 application	50kbs/300ms Orig.	Dol.	%	300kbs/300ms Orig.	Dol.	%	300kbs/50ms Orig.	Dol.	%
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 14: Background code loading overhead. Orig. is the original download time, Dol. is the time to download the whole application in the background, both measured in seconds, and % is the percentage difference.

Web 2.0 applications [13], 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 [8]. 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 a generalization 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.

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 [23, 24]. 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.

Related work includes research into automatic partitioning of programs to execute in a distributed environment [5, 22]. DOLOTO shares many similarities with these tools, such as the automatic insertion of proxy references. The key difference is that all code partitioned by DOLOTO eventually executes on a single machine, and the optimization is to minimize the execution delay due to critical-path code transfer time, not data communication between distributed components.

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# [10, 6, 3]. 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 [18]. 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 [16, 26, 7]. In contrast, DOLOTO uses code rewriting for optimization.

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 directions below.

Static analysis to remove guards: currently, our approach described in Section 3.3 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.

Compiler integration: We hope that the ideas of DOLOTO will be integrated in the next generation of distributing compilers such as Silverlight, Volta, and GWT [12, 11, 4]. Being within a compiler will not only enable static analyses described above, it will also make DOLOTO-like code rewriting part of profile-driven code generation and the runtime environment instead of a separate deployment-time tool.

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 deployed on the server, applications rewritten with DOLOTO exhibit 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 user.

We believe that techniques explored in this paper will become more and more important as distributed Web applications grow in size and complexity, while being increasingly deployed over GSM and 3G networks. Techniques such as Doloto would reduce the need for both geolocation of costly data centers to perform well in an international setting as well as maintaining parallel, slimmed-down versions of existing applications to run on mobile devices.

Moreover, while this paper presents our experiments on JavaScript-based applications, DOLOTO can be seen as a general dynamic code loading approach for distributed applications running on a variety of emerging runtime platforms such as Silverlight and AIR. It is our hope that the code analysis and rewriting techniques proposed in this paper will be integrated in the next generation of distributing compilers such as Volta and GWT, thus making the profile-drive optimizations that DOLOTO implements an integral part of the application development and deployment process.

8. REFERENCES

[1] A. V. Aho, M. Lam, R. Sethi, and J. 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] G. Hunt and M. Scott. Coign automatic distributed partitioning system. In *Proceedings of the Symposium on Operating System Design and Implementation*, 1999.

[6] T. Jensen, D. L. 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.

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

[8] C. Krintz, B. Calder, and U. Hölzle. Reducing transfer delay using Java class file splitting and prefetching. In *OOPSLA*, pages 276–291, 1999.

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

[10] S. Liang and G. 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.

[11] Microsoft Corporation. Microsoft live labs volta. `http://labs.live.com/volta/`, 2007.

[12] Microsoft Corporation. Silverlight. `http://silverlight.net`, 2007.

[13] D. S. Myers, J. N. Carlisle, J. A. Cowling, and B. H. Liskov. MapJAX: Data structure abstractions for asynchronous Web applications. In *Proceedings of the Usenix Technical Conference*, June 2007.

[14] G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance management for cluster based Web services. Technical report, IBM Research, 2003.

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

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

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

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

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

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

[21] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. A scalable application placement controller enterprise data centers. In *WWW 2007: Track: Performance and Scalability*, May 2007.

[22] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic Java application partitioning. In *Proceedings of the European Conference on Object-Oriented Programming*, June 2002.

[23] F. Tip, P. F. Sweeney, and C. Laffra. Extracting library-based Java applications. *Commun. ACM*, 46(8):35–40, 2003.

[24] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter. Practical extraction techniques for Java. *ACM Transactions of Programming Languages and Systems*, 24(6):625–666, 2002.

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

[26] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proceedings of the Conference on the Principle of Programming Languages*, Jan. 2007.